

硕士学位论文

支持频繁更新的 Flash 存储管理技术研究

**RESEARCH ON THE FLASH STORAGE  
MANAGEMENT TECHNIQUE FOR  
THE UPDATE INTENSIVE ENVIRONMENT**

罗永明

哈尔滨工业大学

2009 年 6 月

国内图书分类号：TP311.132  
国际图书分类号：004.6

学校代码：10213  
密级：公开

## 工学硕士学位论文

# 支持频繁更新的 Flash 存储管理技术研究

硕士研究生： 罗永明  
导 师： 李建中 教授  
申请学位级别： 工学硕士  
学 科、专 业： 计算机科学与技术  
所 在 单 位： 计算机科学与技术学院  
答 辩 日 期： 2009 年 6 月  
授 予 学 位 单 位： 哈尔滨工业大学

Classified Index: TP311.132

U.D.C: 004.6

Dissertation for the Master Degree in Engineering.

**RESEARCH ON THE FLASH STORAGE  
MANAGEMENT TECHNIQUE FOR  
THE UPDATE INTENSIVE ENVIRONMENT**

<b>Candidate:</b>	Luo Yongming
<b>Supervisor:</b>	Professor Li Jianzhong
<b>Academic Degree Applied for:</b>	Master of Engineering
<b>Speciality:</b>	Computer Science and Technology
<b>Affiliation:</b>	School of Computer Science and Technology
<b>Date of Defence:</b>	June, 2009
<b>Degree-Conferring-Institution:</b>	Harbin Institute of Technology

## 摘要

随着 Flash 产业的发展与成熟，Flash 存储器作为一种新的存储介质已经被广泛应用到计算机系统中，并有全面取代磁盘的趋势。由于与传统磁盘的读写特性不同，Flash 存储器上的数据管理问题近期成为数据库领域的研究热点。本文集中探讨在数据频繁更新环境下 Flash 存储器上的数据管理问题。

本文提出一种称为 FBX-Tree 的索引结构。该结构在逻辑上提供与 B-Tree 相同的接口，在实际操作文件时将所有对文件的内部更新操作转化为对文件的追加操作，充分利用了 Flash 存储器随机访问文件块较快、更新内部文件块较慢的特点，使系统性能获得提升。FBX-Tree 中的每个叶子节点都在内存中对应一个文件块指针链表，链表长度影响了查询性能，因此需要在适当时刻对其进行整理。我们将决策整理链表时机的问题进行了形式化定义，提出三种近似 on-line 算法对其进行处理，并证明了它们的近似比分别为  $K$ 、3 和  $\frac{\sqrt{K}}{2} + \frac{1}{3}$ ，其中  $K$  为算法给定的阈值。

为说明 FBX-Tree 的使用方法，本文应用其解决移动对象的查询处理问题。本文提出一种基于 FBX-Tree 的移动对象索引方法，该方法使用 FBX-Tree 作为后端存储结构，使用  $B^x$ -Tree 作为前端查询处理方法，支持区域查询、 $K$  近邻查询和持续区域查询。三种查询操作均对 FBX-Tree 做了优化。

基于上述方法本文进行了大量的实验。理论分析和实验结果表明，FBX-Tree 在系统吞吐率和查询性能上均较以往方法有较大改善，并在内存与外存的空间占用上取得了平衡，延长了器件寿命。

**关键词：**Flash 存储器；存储系统；时空索引；移动对象数据库

## Abstract

With the advance in flash industry, as a new type of storage device, flash memory has become widely used in computer system, and will totally replace the magnetic disks in the near future. Because of its different I/O pattern, the data management issues over flash disk become a hot spot in database research. This paper mainly focuses on the data management problem in an update intensive environment over flash disk.

An index structure called FBX-Tree is proposed in this paper. This structure share the same operating interface as B-Tree, meanwhile transform all the internal updates to the append operations to the data file, which makes use of the fast random read and avoids the bad random write performance. Every leaf node in FBX-Tree has a block pointer link list in memory. The length of the list will affect the query performance of the system. In this paper, we formalized this problem and propose three approximate on-line algorithms for it, then we prove that the competitive ratios of the three algorithms are  $K$ ,  $3$  and  $\frac{\sqrt{K}}{2} + \frac{1}{3}$ , in which  $K$  is the given threshold.

To illustrate FBX-Tree's usage, we apply it to solve the query processing problems in moving object database. We propose a moving object indexing method based on FBX-Tree. This method use FBX-Tree as its backend storage system, and use Bx-Tree as its query processing method. Range query, KNN query and Continual Range Query are supported by the system, which are all optimized for the FBX-Tree.

Based on the above methods we do extensive experiments. Theoretical analysis and experiments results show that FBX-Tree improves the system throughput and query efficiency, makes a balance between the primary storage and secondary storage, and extends the device's lifetime.

**Keywords :** flash memory, storage system, spatial index, moving object database

## 目录

摘要 .....	I
Abstract .....	II
<b>第 1 章 绪论</b> .....	<b>1</b>
1.1 研究背景 .....	1
1.1.1 Flash 芯片及存储器特性 .....	1
1.2 Flash 存储管理国内外研究现状 .....	4
1.2.1 基于 Flash 存储器的存储结构 .....	4
1.2.2 基于 Flash 存储器的数据操作 .....	6
1.3 本文的主要研究工作 .....	7
1.4 本文的结构 .....	8
<b>第 2 章 预备知识</b> .....	<b>9</b>
2.1 BFTL .....	9
2.2 空间填充曲线 .....	12
2.3 Bx-Tree .....	13
2.4 本文使用的代价分析符号 .....	16
2.5 本章小结 .....	16
<b>第 3 章 FBX-Tree 索引的构造及维护</b> .....	<b>17</b>
3.1 FBX-Tree 的索引结构 .....	17
3.2 FBX-Tree 的维护 .....	19
3.3 文件块指针链表维护策略 .....	22
3.3.1 问题的提出 .....	22
3.3.2 三种解决方法 .....	26
3.4 本章小结 .....	31
<b>第 4 章 基于 FBX-Tree 的移动对象查询处理方法</b> .....	<b>32</b>
4.1 移动对象数据库原型系统设计 .....	32
4.2 区域查询 (Rang Query) .....	34
4.3 K 近邻查询 (K Nearest Neighbor Query) .....	35
4.4 持续区域查询 (Continual Range Query) .....	36
4.5 本章小结 .....	39

第 5 章 实验结果与分析 .....	41
5.1 实验与分析 .....	41
5.1.1 B-Tree 与 FBX-Tree 性能对比 .....	41
5.1.2 工作负载对系统性能的影响 .....	42
5.1.3 文件块大小的选取 .....	42
5.1.4 日志整理策略的选取 .....	43
5.1.5 日志缓冲区大小对查询性能的影响 .....	44
5.2 本章小结 .....	45
结论 .....	46
参考文献 .....	47
哈尔滨工业大学硕士学位论文原创性声明 .....	52
哈尔滨工业大学硕士学位论文使用授权书 .....	52
致谢 .....	53

# 第 1 章 绪论

## 1.1 研究背景

近年来，闪存（Flash Memory）作为一种存储介质已经得到越来越广泛的使用。最初由于其小巧、省电、防震、低噪音等特点，闪存被应用于 PDA、MP3、手机等便携式设备上。近两年，越来越多的计算机厂商开始使用闪存存储器作为其计算机系统的二级存储设备。Intel 等制造商已经开始涉足量产由闪存组成的阵列存储设备以提供给公司进行企业级应用，一些著名的搜索引擎服务商（google, baidu 等）也开始使用闪存存储器搭建其页面服务器，加速页面响应性能。随着闪存芯片制造业技术的不断成熟及整个市场的不断扩大，Flash 存储器必将向着容量更大、性能更好、价格更低的方向发展。可以预见在不久的将来 Flash 存储器将全面替代磁盘成为计算机系统中的标准配置。然而 Flash 存储器的读写特性与传统磁盘不尽相同，一些根植于 Flash 芯片的固有特性使其在替代磁盘的时候会产生相应的问题。下面就对这些特性做一个初步的介绍。

### 1.1.1 Flash 芯片及存储器特性

Flash 芯片按其构造可以分为两种：MLC（Multi-Level-Cell）和 SLC（Single-Level-Cell）。通常情况下，MLC 具有更大的存储密度和更低廉的制造成本，相应有着更长的存取时间。目前用于存储大规模数据的芯片均为 MLC。

Flash 存储芯片是 Flash 存储设备的基础，其在存储器内部级联工作，逻辑组成见图 1-1。Flash 芯片对数据的读写单位是一个页（page），多个相邻页组成一个擦除单元（erase unit）。在向一个已有数据的页中写入数据时需要先对其所属的擦除单元进行擦除操作。对一个擦除单元进行一定次数（ $10^4 \sim 10^5$ ）的擦除操作后，整个擦除单元将失效（wear out），所以除了存储正常的数据外，一个擦除单元中还要预留一部分空间用于存储维护信息，比如擦除次数等。

Flash 芯片具有以下性质：

- 无机械延迟。由于传统磁盘的机械结构限制，其读取一个磁盘块的时间通常由寻道时间、旋转等待时间和数据传输时间几部分构成。而对于 Flash 这种纯电子器件来说，不存在寻道时间和旋转等待时间，与之对应的是译码等计算

时间，这使得其随机读取任意给定地址文件块的时间基本上是一致的，在寻址时间延迟上远少于磁盘。

- 读写速度不一致。由于器件本身的构造特性，Flash 芯片的读写速度是不一致的，通常情况下读比写要快。对于不同的 flash 芯片设备而言，读写速度比通常在 2 到 100 左右。
- 无法对数据页进行原地更新。这里的原地更新，是指在数据的原存储位置进行覆盖写操作。Flash 存储器无法进行原地更新，在更新本页前需要对整个页所属的擦除单元进行擦除操作。
- 只能进行有限次擦除操作。组成 Flash 芯片的每个擦除单元都是有使用寿命的，在达到擦除次数上限后，芯片将无法正常工作。这使得在进行擦除操作时，必须考虑到使得各擦除单元的擦除次数大致相当，尽量不造成某块提前实效，并且尽可能少进行擦除操作，延长器件使用寿命。这种技术称为损耗平衡技术（wear leveling）<sup>[1]</sup>，通常由存储器内部的控制器进行协调控制。

表 1-1 Flash 芯片性能参数<sup>[2]</sup>

芯片指标	参数
Page Read to Register	25 μ s
Page Program (Write) from Register	200 μ s
Block Erase	1.5ms
Serial Access to Register (Data bus)	100 μ s
Die Size	2 GB
Block Size	256 KB
Page Size	4 KB
Data Register	4 KB
Planes per die	4
Dies per package (2GB/4GB/8GB)	1,2 or 4
Program/Erase Cycles	100 K

表 1-1 给出了某典型 Flash 芯片的性能参数。可以看到将数据从寄存器写入芯片和擦除整个单元的时间都比将一个页读入寄存器的时间长得多。由于更新将触发擦除单元操作，可以预见当对 Flash 更新操作频繁时，大量的擦除单元操作将拖慢系统的整体性能。

SSD（Solid State Drive）<sup>[3]</sup>是目前在计算机上使用最广泛的一种 Flash 存储设备，也是最有可能替代传统磁盘成为新的通用外存储器的设备。该设备通过标准的 IDE 接口或 SATA 接口与计算机进行连接，操作系统操作 SSD 时如同操作普通的磁盘，不需要进行额外配置。SSD 通过 FTL（Flash Translation Layer）层建立其逻辑视图。该层经由硬件或者软件手段将 Flash 数据页地址编址成与传统磁盘相同的扇区号。各厂商的 FTL 层实现不尽相同<sup>[4]</sup>，并且多

为专利信息不与外界共享。

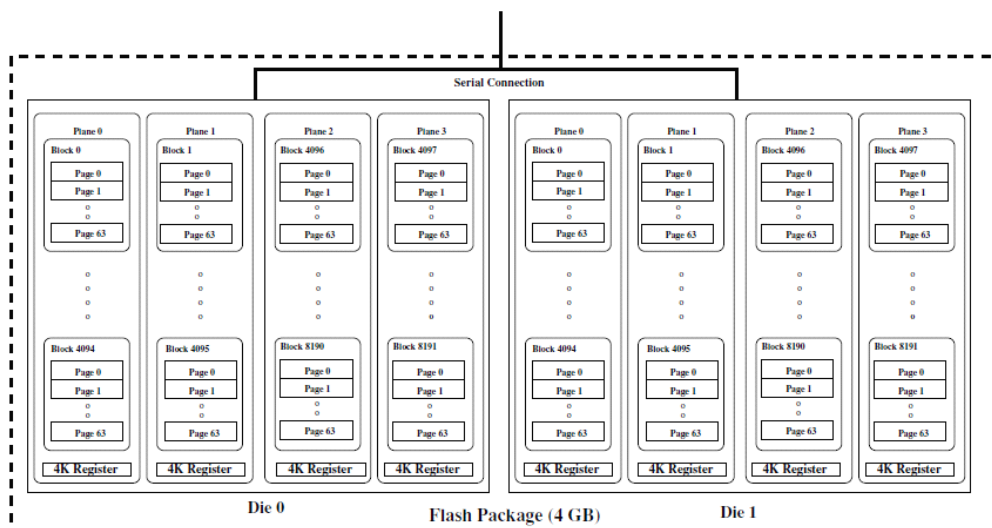


图 1-1 三星 SSD 内部 Flash 芯片组成结构示意图<sup>[2]</sup>

Figure 1-1 Samsung SSD Flash Internal

图 1-1 是一个三星 SSD 存储部分的内部结构图。对于该器件，每 64 个页组成一个擦除单元，每 2048 个擦除单元组成一个控制单元，配备一个读写寄存器。各个控制单元间并行连接后编址，与控制器、RAM 等结合构成一个完整的 SSD。

表 2-2 Flash 存储器与传统磁盘性能对比<sup>[5]</sup>

性能参数	SATA 磁盘	USB Flash	IDE Flash
耗电量 (W)	13	0.5	0.5
顺序读取速度 (MB/s)	60	26	28
顺序写入速度 (MB/s)	55	20	24
随机读取次数 (IO/s)	120	1,500	2,500
随机写入次数 (IO/s)	120	40	20

通过在 SSD 内部设置 RAM 与处理器，SSD 的读写性能较 Flash 芯片已有较大改善，但仍体现出一些与传统磁盘不同的特征。表 2-2 显示了在读写性能参数上 Flash 存储器（U 盘、SSD）与 SATA 磁盘之间的对比。基于这些数据和真实实验，有以下观察：

- 顺序读取与随机读取速度大致相当。这是 Flash 存储器明显优于磁盘的一点。传统磁盘上的算法均基于磁盘顺序读写快于随机读写的假设，在该假设不成立的条件下，一些算法应该被重新设计。

● 对文件的更新操作十分耗时。Flash 芯片本身就有读比写快的性质，由于其页不能原地更新，对某个页的更新操作大致被翻译成将原页读入 RAM，修改后写入新页，旧页等待擦除这一系列命令，比 Flash 芯片的写操作更加耗时。根据表格 2 及实际试验，在 Flash 存储器上单位时间内的随机读写速度比大约为 10:1 至 100:1，因此以往的一些制造较多文件更新操作的外存算法将不再适用于 Flash 存储器。

## 1.2 Flash 存储管理国内外研究现状

由于 Flash 存储器具有以上特性，近一段时间以来，Flash 存储器上的数据管理问题成为数据库领域的研究热点。目前的研究工作主要集中在在 Flash 上进行存储结构和数据操作的设计。由于 Flash 存储器上最耗时的操作为页更新操作，故已有的解决方法的大体思路都是减少更新页的操作，或者以大量读操作代替更新操作。

### 1.2.1 基于 Flash 存储器的存储结构

B-Tree 是文件系统和数据库系统中最常用的一种数据结构，所以针对 B-Tree 的研究也较早开始。B-Tree 是一种基于区域划分的平衡搜索树，每个树节点一般被设定为一个数据页的大小。当直接在 Flash 存储器上建立 B-Tree 索引后，每更新一个叶子节点就要更新一个文件块，这使得在一个数据频繁更新的环境里，在 Flash 存储器上使用 B-Tree 性能较差。

为减少经典的存储结构受 flash 读写特性的不良影响，以往文献提出了一些解决方案<sup>[6-16]</sup>。它们的主要思想为将更新操作转化为对文件块修改的日志操作追加到文件末端。文献[6]提出了在 flash 上实现 B-Tree 的方法。它在内存中维护一个节点映射表（NTT），所有针对节点的更新操作作为日志追加到系统中。在 NTT 中每个节点头串起一系列包含当前节点日志项的页号，当系统需要读取某一个 B-Tree 节点时，通过读取这一系列数据页重建该节点。该方法降低了写代价（更新操作代价由写日志页分担），提高了读代价，而且由于系统对节点头指向的链表长度缺乏管理策略，系统的查询性能无法得到保证。在 2.1 节将对这种结构做详细的介绍。文献[7]构造了 Flash 存储器上的 R-Tree 索引，其系统框架结构、维护规则与文献[6]完全相同。

文献[8]基于文献[6]进行了更深入的探讨。本文的作者认为系统本身的结构应随系统的读写负载变化进行调整，以获得更好的性能。本文中的 B-Tree 节点具有两种状态：日志状态和磁盘状态。在日志状态下，页中的每个元组与[6]中相同，物理上为追加到磁盘上的分散的日志，由内存中维护节点映射表

来保持其逻辑结构，利用日志操作提升写入性能。在磁盘状态下，系统将 Flash 存储器中的文件块视为传统的磁盘块进行操作，这就保证了系统可以以较快的速度读取磁盘状态页。每个节点在这两种状态间进行转换，当频繁读某一节点时，其转换到磁盘模式，当频繁更新某一节点时，其转变为日志模式。对于一个已知的操作序列，存在一个最优的状态转换序列，但由于操作序列不可知，系统无法达到最优状态。本文作者设计了一种 on-line 算法对节点状态转换进行决策，并证明了该 on-line 算法的近似比为 3。这种节点的状态转换策略也可以应用于其他的块结构系统中。

文献[9]在[8]的工作上更进一步，该文章探讨的是当系统中同时使用磁盘和 Flash 存储器进行工作时，如何进行存储管理。由于磁盘拥有较好的写特性，而 Flash 存储器拥有较好的读特性，可以想象如果将写操作较多的页放在磁盘上，将读操作较多的页放在 Flash 存储器上，系统的性能应该会达到最优。作者设计了一个与文献[8]类似的 on-line 算法，为每个页设计两种状态：磁盘态和 Flash 态，并统计每个页的读写次数，当到某个页的读次数或写次数到达一定阈值后，便转换到另一个状态，即转移到另一个存储介质上。

文献[10]用一棵修正的搜索树来为数据建立索引。文章提出将页组织成一个个段（比如一个擦除单元的大小），段中的页分为索引页和数据页，数据页按时间顺序依次追加直至一个段满，索引页维护两个到下级的指针和当前域值指向的数据，这样每次插入一条记录就不会造成多个更新，如果索引页满，会触发下一级索引页继续使用。这种方法降低了写代价，但使读代价变得异常高，每次执行一个查询，不仅可能要在一个段中搜索多个索引页，还有可能搜索多个段，极端情况下需要遍历所有段。

文献[11]提出一种与具体数据结构无关的日志策略。该策略以擦除单元为单位管理数据，为每个擦除单元分配固定数量的日志页，用于记录本擦除单元内的页更新日志。这使得一个页的相关日志集中于其所在的擦除单元中，而不会散落在整个物理空间，而且因为日志的配额固定大小，某个页的相关日志不会变得过长，从总体上保证了一定的读写性能。

文献[12]提出了一种类似 B-Tree 的小型数据结构  $\mu$ -Tree。在  $\mu$ -Tree 中，节点由叶子到根容量逐渐减小，每一条由叶子到根的路径都可以放到一个数据页中，这就保证了当更新一个叶子时所有需要更新的各级节点都写入到一个数据页中，旧数据页失效等待回收。

文献[13]也提出了一种支持搜索的结构，在该结构中，分别建立数据区，一级索引区、二级索引区等结构，数据区内的数据顺序增长，索引区中存放数

据区中每个文件块的摘要信息。在索引区中，使用 Bloom Filter 对摘要信息进行二次过滤，如此使得系统的查询性能得到保证。

文献[14]提出了一种对查询进行了优化的存储结构 FD-Tree。系统中维护着多级线性存储结构，当数据到来时，首先在内存中进行索引，索引装满后将其与下一级索引进行合并，该操作递归进行。该结构保证了查询的快速处理，但在进行合并操作时有可能造成系统的长时间停顿，影响系统的响应效果。

文献[15]和文献[16]探讨了在 flash 存储器上建立 Hash 索引的问题。其中文献[15]针对传感器节点中的 flash 芯片存储进行设计，不能被应用在其他情况下。文献[16]是一种通用的解决方案，它基于线性 hash 修改。其通过推迟或提前线性 hash 的节点分裂时间，使得其拥有更好的性能。其包含两种策略：EagerSchema 大致与线性 hash 相同，只是在分裂后再插入新的数据；Lazy schema 缓存当前的分裂操作，当其对搜索性能影响过大即达到某一阈值时进行批量分裂。

以上方法中，文献[8,10,12,13,15]为针对嵌入式系统进行的设计，其代价模型与 SSD 不尽相同，且数据量较小，应用于计算机系统中后性能无法得到保证；文献[11]的方法无法在操作系统以上的层面实现。且上述方法均未考虑在数据频繁更新的环境中如何使系统保持较高的吞吐率与查询性能。

### 1.2.2 基于 Flash 存储器的数据操作

文献[5]第一次探讨了在 Flash 存储器上做 Join 操作的问题。作者将 Flash 存储器的物理页称为小页，其设计的 Join 操作逻辑页中包含多个小页。元组在逻辑页中按列组织起来（见图 1-2），这样指定属性就落在有限几个小页中。如此系统做 Join 操作的时候只读取相应的小页即可得到结果，而不必扫描所有逻辑页。基于 Flash 存储器定位文件块时间特别短的优势，操作得到了更好的执行性能，有效利用了 Flash 存储器随机读取速度快的特性。文献[17]研究了如何利用 flash 读写特性在大吞吐率数据流上进行随机采样操作的问题。

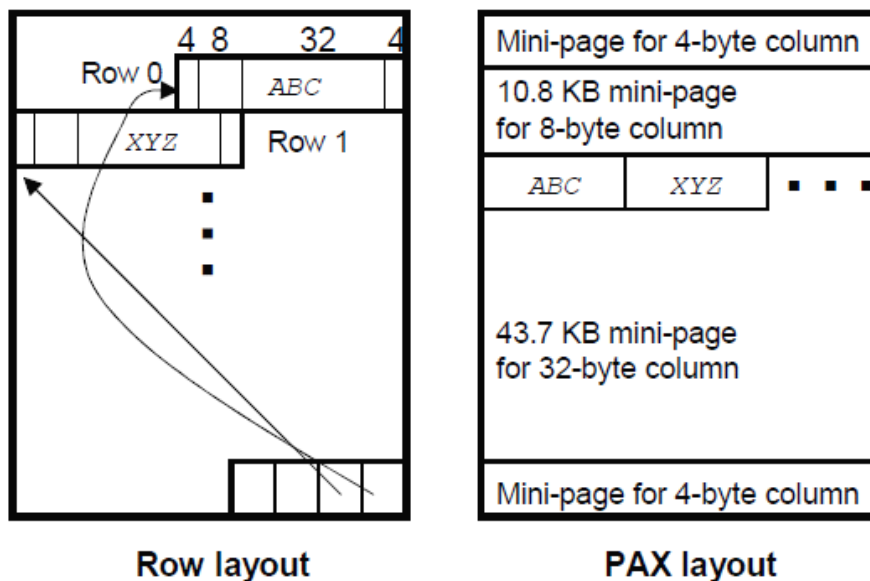


图 1-2 按列存储进行 Join 操作<sup>[5]</sup>

Figure 1-2 Using PAX Layout do the Join Operation

### 1.3 本文的主要研究工作

本文主要考虑在数据频繁更新的环境下如何使用 Flash 存储器有效的进行数据管理。目前针对该问题的的工作还比较少。直接在数据频繁更新的环境中使用 Flash 存储器有以下问题：（1）由于 Flash 存储器对于文件的内部更新操作表现较差，传统的外存算法在 flash 上运行无法获得良好的性能。（2）大量的更新操作会导致 flash 器件寿命显著缩短。

本文提出了一种称为 FBX-Tree 的索引方案。该方案通过日志信息和缓存操作将所有的数据更新转化为对数据文件的顺序写操作，有效避免了对文件内部进行更新，保证了系统的吞吐率并延长了器件寿命。通过两种 on-line 算法对内存中的文件块指针链表进行维护，系统的查询性能得到保证。

本文的主要有以下贡献：

- 设计了一个 flash 存储器下的针对数据频繁更新的索引方案 FBX-Tree。
- 提出了三种整理内存中文件块指针链表长度的 on-line 算法，并分析了它们的近似比。
- 利用 FBX-Tree 作为存储后端，B<sup>x</sup>-Tree 作为查询处理前端，设计了 Flash 存储器上的移动对象索引结构。该结构支持区域查询、K 近邻查询和持续区域查询。所有查询都针对 FBX-Tree 进行了优化。
- 通过大量的实验验证了以上方法的有效性。

## 1.4 本文的结构

本文分为六个部分。

第 1 章是绪论。本章简要介绍有关 Flash 存储器的基本知识，国内外对于 Flash 存储器上的数据管理问题研究现状，以及本文的主要研究工作。

第 2 章是预备知识。本章详细介绍 Flash 上的 B-Tree 实现 BFLL，空间填充曲线的性质和用途，以及基于 B-Tree 的移动对象索引方案 B<sup>x</sup>-Tree。

第 3 章介绍本文提出的索引方案 FBX-Tree，详细说明 FBX-Tree 的构造方法和维护策略。对维护过程中出现的文件块指针链表过长问题，提出三种 on-line 算法进行解决，并证明了他们的近似比。

第 4 章主要设计基于 FBX-Tree 和 B<sup>x</sup>-Tree 的移动对象索引方法，该方法支持区域查询、K 近邻查询和持续区域查询。以上查询均针对 FBX-Tree 特点进行优化。

第 5 章基于上述方法进行大量实验。实验结果表明 FBX-Tree 在保持高效吞吐率和查询效率的同时，占用了较少的内存和外存空间，延长了存储器使用寿命。

最后对本文的工作及贡献进行总结，并指出需要进一步研究的问题。

## 第 2 章 预备知识

本章主要介绍论文中一些方法的预备知识。2.1 节介绍 Flash 下的 B-Tree 实现方案 BFTL，包括其构造方法和维护策略；2.2 节介绍空间填充曲线的特性和用途；2.3 节详细介绍基于 B-Tree 的移动对象索引方案 B<sup>x</sup>-Tree；2.4 节对后文使用的代价分析符号进行约定。

### 2.1 BFTL

BFTL<sup>[6]</sup>是一种在 Flash 存储器上实现 B-Tree 索引的方案。其在操作系统的 FTL 层和应用软件逻辑层之间加入 BFTL 层，对于上层应用来说，它具有 B-Tree 的所有操作接口，对下层文件系统来说，它仅制造对外存数据文件的追加操作。

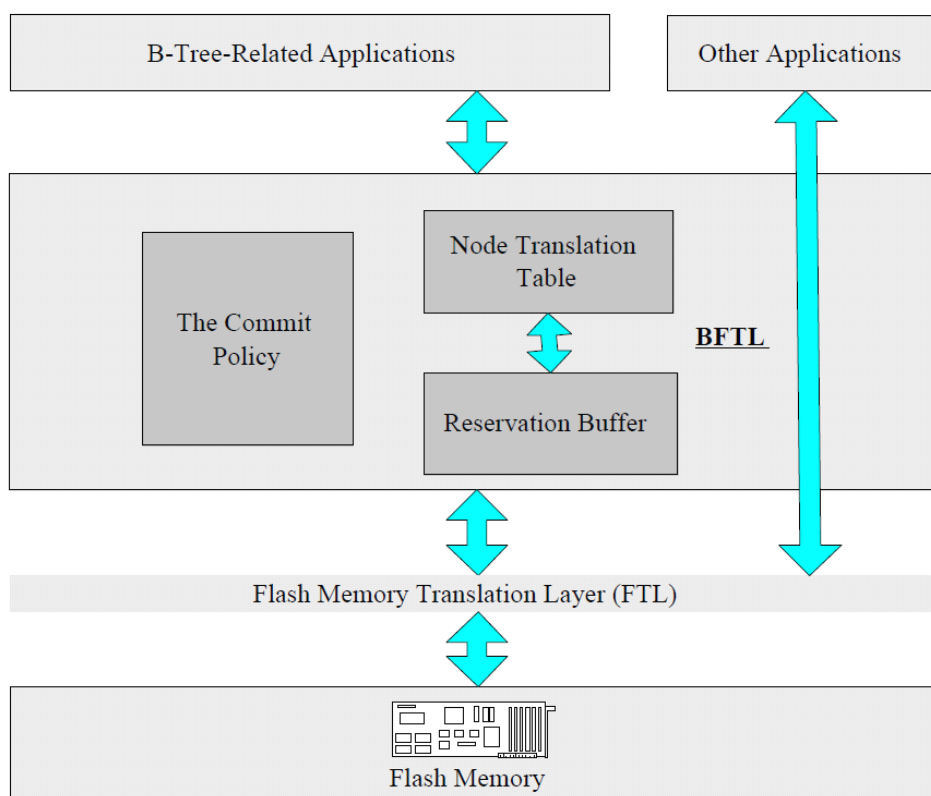


图 2-1 BFTL 框架结构<sup>[6]</sup>

Figure 2-1 The Framework of BFTL

BFTL 框架结构如图 2-1 所示，在内存中维护节点转换表（Node Translation Table）和日志缓冲区（Reservation Buffer），并由控制策略

(Commit Policy) 调配两个结构的工作。对 BFTL 内的每个节点操作转化为一个索引单元 (index unit)，其内容包括键值，记录的数据，所属的节点，对应操作等等。系统工作时，数据转换为索引单元后进入日志缓冲区，当日志缓冲区充满时追加到外存的文件末端。同时，在节点转换表中相关的节点需要添加一个指向该文件块位置的指针。系统需要读入某节点时，通过节点转换表获得所有与其相关的文件块指针，依次读入这些文件块，根据文件块中的索引单元重建该节点。

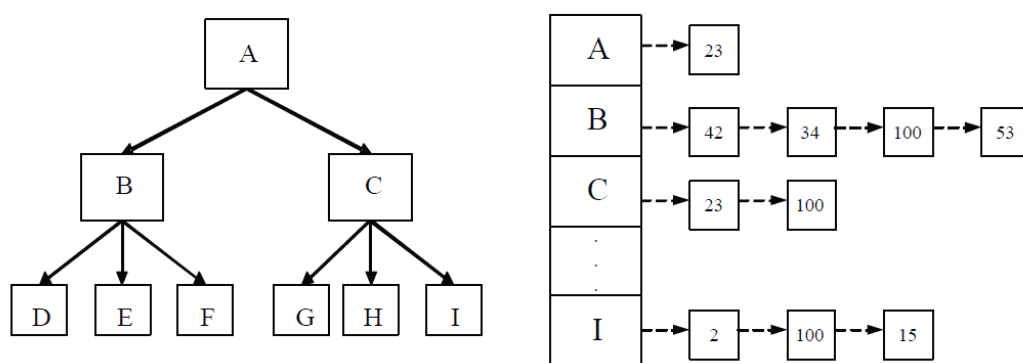


图 2-2 BFTL 逻辑视图及其对应节点转换表举例

Figure 2-2 Example for Logical View and the Node Translation Table

图 2-2 展示了某棵 B-Tree 的逻辑视图及其对应的节点转换表的构造。在内存中为每个节点建立对应区域，将与该节点相关的文件块指针添加到节点后的链表中。如图中的节点 B，需要读入块 42，34，100，53 才能将其还原。

BFTL 执行搜索操作时，首先检查符合条件的记录是否在缓冲区中，若在缓冲区中则返回该记录。若不在缓冲区中，则对逻辑 B-Tree 进行搜索操作，并返回搜索结果。

#### 算法 2-1 BFTL 搜索算法

算法的输入：键值 K。

算法的输出：搜索结果，返回 NULL 如果没有找到具有键值 K 的记录。

01. 在缓冲区中检查是否有相关记录；
02. If 找到记录
03. 返回记录；
04. Else //搜索逻辑 B-Tree
05. 从根节点到叶子节点
06. 重建路径上的每个节点；
07. 使用 B-Tree 搜索策略搜索该节点；

08. If 找到记录
09. 返回记录;
- 10.返回 NULL;

向 BFTL 插入记录时, 首先需要执行搜索操作找到该记录所属的叶子节点, 找到后为记录生成索引单元将其放入日志缓冲区中。若缓冲区满则将其写到文件末端。

#### 算法 2-2 BFTL 插入算法

算法的输入: 记录 R。

算法的输出: 插入结果 (成功或者失败)。

- 01.从根节点到叶子节点的父亲节点
02. 重建路径上的每个节点;
03. 使用 R 的键值搜索节点;
04. 得到叶子节点号;
- 05.为记录 R 准备一个索引单元;
- 06.将该索引单元插入缓冲区;
- 07.If 缓冲区已满
08. 将缓冲区文件块追加到数据文件;
09. 更新相关的节点转换表入口;

BFTL 中的删除操作与插入操作类似, 首先检查符合条件的记录是否在缓冲区中, 若在缓冲区中则直接删除该记录。若不在缓冲区中, 则对 BFTL 做搜索操作。若找到记录, 则生成具有删除操作的索引单元; 若没找到记录, 则无其他动作, 直接返回。

#### 算法 2-3 BTFL 删除算法

算法的输入: 键值 K。

算法的输出: 删除结果 (True 或者 False)。

- 01.检查记录是否在缓冲区中;
- 02.If 找到记录
03. 删除该记录;
04. 返回 True ;
- 05.Else
06. 从根节点到叶子节点
07. 重建路径上的每个节点;
08. 使用键值 K 搜索节点;

09. If 找到拥有键值  $K$  的记录  $R$
10.     使用  $K$  和  $R$  构造删除索引单元；
11.     插入索引单元到缓冲区；
12.     If 缓冲区已满
13.         将缓冲区文件块追加到数据文件；
14.         更新相关的节点转换表入口；
15.     返回 True；
16. Else
17.     返回 False；

## 2.2 空间填充曲线

空间填充曲线（space filling curve）<sup>[30]</sup>是对多维空间进行降维操作时经常使用的一种方法。通过空间填充曲线，多维空间中的数据点在降维后依然保留一定的聚集特性，即在高维空间中距离较近的点在降维后依然距离较近。这样就可以使用低维空间上的数据处理方法（索引、采样等）对高维数据进行处理。常用的空间填充曲线有 Hilbert 曲线（Hilbert Curve）、Z 排序曲线（Z-Order Curve）等。根据文献[31]的说明，对于时空数据，Hilbert 曲线的聚集特性要好于 Z 排序曲线，所以我们下面针对 Hilbert 曲线进行探讨。

空间填充曲线具有如下的拓扑性质：若将空间划分为相同大小的网格，曲线可以按照一定的规则遍历每个网格一次，把空间填满，且曲线本身无任何交叉。按照曲线遍历网格的次序可以为每个网格创建唯一的编号。

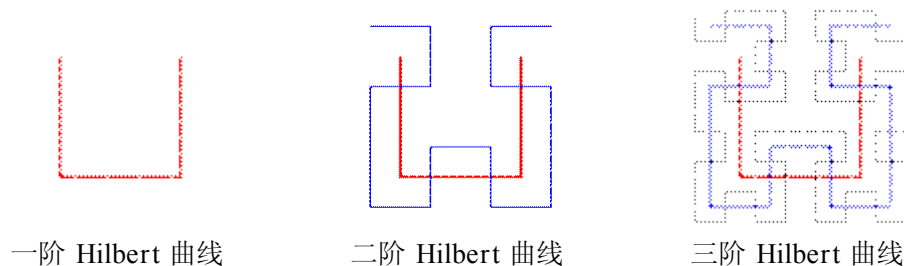


图 2-3 Hilbert 曲线举例<sup>[32]</sup>

Figure2-3 Example of Hilbert Curve<sup>[32]</sup>

Hilbert 曲线可以递归的画出：一阶的 Hilbert 曲线如 U 字形，可以填充一个  $2 \times 2$  的网格；阶数为  $K$  的 Hilbert 曲线可以由  $K-1$  阶曲线得到，具体为对  $K$  阶曲线的每个顶点（包括端点），用一阶曲线替换，并进行必要的连接。替换时需要根据规则进行必要的旋转。图 2-3 分别展示了一阶、二阶和三阶的 Hilbert 曲线，及其怎样由低阶曲线得到。

曲线的阶数越高，对空间划分出的网格粒度越小，当阶数足够高时，空间中的每个网格将只包含不超过一个数据对象，整个空间将被曲线填满。在第4章方案中，曲线阶数最大为10，即网格数为 $2^{10} \times 2^{10}$ 。若用二进制编码表示Hilbert填充的网格编码，可以发现，拥有相同前缀越长，两个编码区域距离越近。利用这种性质可以对网格中的数据进行聚集。

### 2.3 B<sup>x</sup>-Tree

移动对象数据库<sup>[33-35]</sup>是一种需要频繁进行数据更新的环境。基于传统的磁盘代价模型，近几年已经提出了大量的索引结构，主要分为索引过去与当前位置，索引当前与将来位置两大类。基于划分方式的不同，针对移动对象的索引方法还可以分为基于空间的划分（Grid File<sup>[36]</sup>，B<sup>x</sup>-Tree<sup>[37]</sup>，B<sup>dual</sup>-Tree<sup>[38]</sup>，ST<sup>2</sup>B-Tree<sup>[39]</sup>等）和基于对象的划分（R\*-Tree<sup>[40]</sup>，Quad Tree<sup>[41]</sup>，TPR\*-Tree<sup>[42]</sup>等）两种。其中B<sup>x</sup>-Tree由于其高效的性能并可以与现有数据库结合在近年来获得了广泛的关注。

B<sup>x</sup>-Tree将B-Tree引入移动对象领域，通过空间填充曲线，将二维空间的移动对象信息转化为一维数据并存储在B-Tree中。在B<sup>x</sup>-Tree中，移动对象数据用一个时间线性函数和更新时刻来描述，在二维空间中具体为 $(\vec{x}, \vec{v}, t_u)$ ，其中 $\vec{x}$ 为物体位置向量， $\vec{v}$ 为物体当前速度向量， $t_s$ 为采集物体数据时刻。如此任一时刻的对象位置都可以由这三个量计算出来。

B<sup>x</sup>-Tree将时间划分为不同的时段（phase）。具体为，定义 $\Delta t_{mu}$ 为任意对象的最大更新时间，根据 $\Delta t_{mu}$ 划分时间轴为不同的区间，则每个区间内每个对象至少更新一次。再将区间等分为 $n$ 个子区间，使子区间大小为对象的最短更新时间，这样每个子区间中对象至多更新一次。这样划分出的每个子区间称为一个时段，对应一棵B-Tree，如此，同一个对象在每棵B-Tree中仅有可能出现一次，不存在对其自身的更新问题。每个时段对应一个时间参考点，在该时段内更新的对象需要将自己的位置计算至时间参考点以待索引。时刻 $t_s$ 的更新计算至时间参考点的方法为 $t_{ref} = \lceil t_s + \Delta t_{mu} / n \rceil$ ，其中 $\lceil x \rceil$ 表示距离时间 $x$ 最近的时间参考点。

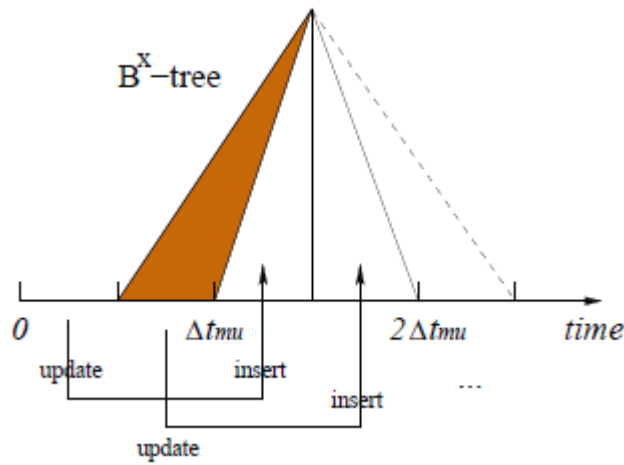


图 2-4 B<sup>x</sup>-Tree 的时段举例<sup>[37]</sup>

Figure2-4 Example of B<sup>x</sup>-Tree Time Phase

图 2-4 所示的为  $n=2$  时的  $bx$ -tree。经计算，在时间戳  $t=0$  时刻的对象对应的时段为  $\Delta t_{mu} / 2$ ，在  $0$  至  $\Delta t_{mu} / 2$  之内的对象对应的时段为  $\Delta t_{mu}$ 。

移动对象在 B<sup>x</sup>-Tree 上索引的键值由时段标号和对象在时间参考点的位置两部分拼接而成。公式表示为：

$$B^x - value(O, t_u) = [phase]_2 \oplus [x - rep]_2 \quad (2-1)$$

这里  $[x]_2$  表示  $x$  的二进制形式， $\oplus$  表示前后两部分做连接操作，公式 (2-1) 的两部分定义如下：

$$phase = (t_{lab} / (\Delta t_{mu} / n) - 1) \bmod (n + 1)$$

$$x - rep = x\_value(\vec{x} + \vec{v} \cdot (t_{lab} - t_u))$$

$x - rep$  可以通过空间填充曲线获得，其中  $\vec{x}$  和  $\vec{v}$  为给定对象的位置向量和速度向量， $t_u$  为对象发生更新的时刻。

$x$ -value 计算举例：设  $n=2$ ， $\Delta t_{mu}=120$ ，空间被划分为  $8 \times 8$  的网格，使用 3 阶曲线填充。有对象  $O_1$ 、 $O_2$ 、 $O_3$  分别对应时间  $0$ ， $10$ ， $100$ ，那么  $x$ -value 的计算方法如下：

步骤 1：计算对象所属的时段和参考时间

$$t_1 = \lceil 0 + 120 / 2 \rceil = 60, \quad phase = 0 = (00)_2$$

$$t_2 = \lceil 10 + 120 / 2 \rceil = 120, \quad phase = 1 = (01)_2$$

$$t_3 = \lceil 100 + 120 / 2 \rceil = 180, \quad phase = 2 = (10)_2$$

步骤 2：计算对象相对参考时间的位置

$$x_1' = (1, 5), x_2' = (2, 3), x_3' = (4, 1)$$

步骤 3: 由曲线计算位置对应的值

$$[H\_value(x_1')]_2=(010011)_2$$

$$[H\_value(x_1')]_2=(001101)_2$$

$$[H\_value(x_1')]_2=(100001)_2$$

步骤 4: 计算 x-value

$$B^xvalue(O_{1,0})=(00010011)_2=19$$

$$B^xvalue(O_{1,0})=(01001101)_2=77$$

$$B^xvalue(O_{1,0})=(10100001)_2=161$$

$B^x$ -Tree 支持区域查询与 K 近邻查询。当查询到来时, 系统首先确定该查询与哪些时段有关, 并将查询分配到相关下层 B-Tree 上执行。进行区域查询操作时, 系统根据当前统计信息和时间参考点扩展查询区域, 然后转换该区域查询为多个一维区域查询操作执行。如图 2-5,  $t_{ref}$  是  $p_1, p_2, p_3, p_4$  的参考时间, 索引时, 其各自被计算到空心圆的位置, 当查询来到时, 需要查询窗口根据对象的速度信息扩大到虚线位置, 得到查询结果为这 4 个对象。

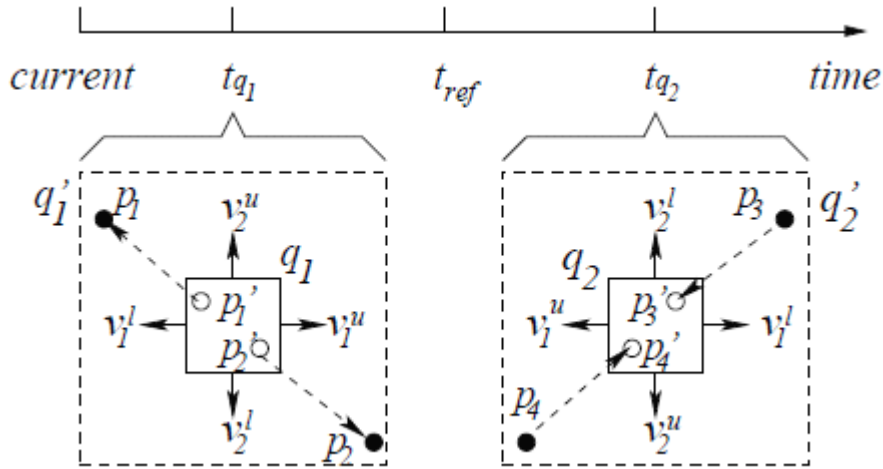


图 2-5 Bx-Tree 的查询窗口扩展<sup>[37]</sup>

Figure2-5 Query Window Enlargement in  $B^x$ -Tree

## 2.4 本文使用的代价分析符号

为方便后文陈述，本文对代价分析有以下符号约定：

表 2-1 代价分析符号约定

符号	意义
$R_{seq}$	顺序读取文件块开销
$R_{rdm}$	随机读取文件块开销
$W_{seq}$	追加写入文件块开销
$W_{rdm}$	随机写入文件块开销
H	逻辑 B-Tree 树高
D	叶子节点度
$L_i$	节点 i 的日志链长度

## 2.5 本章小结

本章主要介绍了论文中所用方法的一些预备知识，包括在 Flash 上建立 B-Tree 索引的工作 BFTL，空间填充曲线的相关知识，使用 B-Tree 作为存储后端的移动对象索引方法 B<sup>x</sup>-Tree，以及一些后文将用到的代价分析符号。

## 第 3 章 FBX-Tree 索引的构造及维护

为克服以往工作在数据频繁更新环境下的不足，本章提出一种新的索引结构 FBX-Tree。在详细介绍该结构的构造方法和维护操作的基础上，本章提出三个 on-line 算法用以解决系统维护过程中出现的文件块指针链表整理问题，并证明它们的近似比。

### 3.1 FBX-Tree 的索引结构

为在数据频繁更新环境使用 Flash 存储器，首先考虑直接使用 2.1 节中提到的 BFTL 结构。BFTL 有以下不足：（1）查询性能无法得到保证。BFTL 中的每个节点都以日志形式存储在文件中，设内存中日志链长度最大为  $L$ ，则进行一次查询操作需要的开销为  $R_{rdm} \times H \times L$ ，且对日志链的维护没有很好的策略。（2）维护性能有待提高。对 BFTL 进行插入删除操作时，需要读入  $H-1$  个内节点，代价为  $R_{rdm} \times (H-1) \times L$ ，日志写入日志缓冲区最终追加至文件末尾的平摊代价为  $W_{seq}/D$ ，性能有很大的提升空间。

BFTL 的优点和不足提供给本文如下设计思路：

- 在运行过程中只对文件进行追加操作。由于 Flash 存储器的读写特性，文件内部的更新操作将严重影响系统性能。故与 BFTL 类似，本方案在系统运行过程中只对文件进行追加操作。
- 加快内节点的读取速度。由于 B-Tree 中无论查询还是插入操作，都要对内节点进行读操作，所以在本方案中考虑加快内节点的读操作性能，以提高系统整体性能。
- 日志链的维护问题。日志链的维护问题在以往工作中没有受到足够的重视。本方案中着重考虑本问题，以给出更好的维护策略，加速查询性能。

基于以上设计思路，本文设计了一种新的索引结构 FBX-Tree。FBX-Tree 在逻辑上为一棵 B-Tree，并可以在数据频繁更新的环境中工作，同时保持高效的查询性能。

FBX-Tree 在内存中维护以下模块：节点转换表，日志缓冲区，节点缓冲区。节点转换表存储每个节点的元数据（节点号，读写次数，节点状态等），并记录节点数据存储的位置信息。日志缓冲区缓冲对叶子节点的操作日志，包括增加记录、删除记录、更新记录。日志信息的格式为（键值，数据，属于节点号，时间戳，操作），操作可为插入、更新和删除。节点缓冲区缓冲已读入的一部分节点，并使用 LRU（Least Recently Used）策略进行替换。

为加速查询和维护性能，FBX-Tree 中的内节点数据不以日志形式存储，而以传统的文件块形式保存在外存中，当系统需要对内节点数据进行修改时（如内节点分裂），系统读入内节点数据，更改后追加一个新文件块到文件末尾。

FBX-Tree 使用日志方法记录对叶子节点的修改，并将与同一叶子节点相关的日志存放在不同文件块中，由节点映射表中的文件块指针链表记录各个块号。当系统需要获得某个叶子节点时，需要读入指针链表所指向的所有文件块，以重建节点。

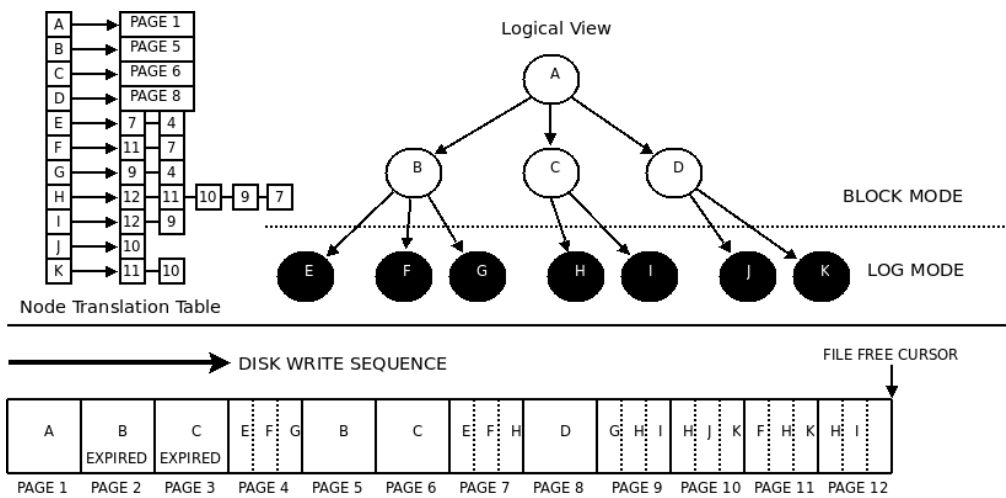


图 3-1 FBX-Tree 构造

Figure 3-1 Construct of FBX-Tree

图 3-1 为构造 FBX-Tree 的一个例子，其逻辑视图为一棵 B-Tree。其中的 A 至 D 为内节点，E 至 K 为叶子节点。节点度为 3，日志缓冲区大小为一个文件块大小，可以存放三条日志记录。随着树的生长，节点信息不断向文件末端添加。对于节点 A，其节点映射表中指向页 1，表示其信息存储在文件块 1 中。对于节点 B，其节点映射表本指向页 2，当信息修改后，改为指向页 5，同时页 2 被标记为失效。节点 H 为叶子节点，其指向链表(12,11,10,9,7)，表示节点 H 的信息分散在这些块号的文件块中。

FBX-Tree 对 Flash 存储器的所有写操作均为文件追加操作。对于叶子节点的更新操作首先出现在日志缓冲区中，然后被追加到文件末端。对于内节点的更新操作也为对文件的追加操作，新文件块即为更新后的内节点。

随着操作的不断进行，外存中的数据文件持续增长，这造成系统无法对文件前端部分的一些失效文件块进行回收再利用，浪费了存储空间。在 FBX-Tree 中，每隔一段时间系统建立新的数据文件，在新的文件建立后，系统只

对新文件进行追加操作，旧文件只用于读取节点信息。当某一文件中所有文件块都被标记为失效后，系统将其删除，数据空间得到回收。这种情况下，在日志节点的文件块链表中，不仅要记录文件内部的偏移地址，还要记录该文件的句柄以进行区分。

### 3.2 FBX-Tree 的维护

本节介绍 FBX-Tree 的各种维护操作，包括重建节点操作，插入操作，删除操作和节点分裂操作。对于每种操作都介绍了其工作过程、具体算法和代价。

重建节点操作为 FBX-Tree 中的基础操作，其功能为从 FBX-Tree 中得到指定节点。操作执行时分为三种情况考虑，分别为当节点在缓冲区中时，节点为内节点时和节点为叶子节点时。对于一个处于节点缓冲区中的节点，系统直接得到其内存指针。对于处于外存上的内节点，系统需要读取一个文件块，故开销为  $R_{rdm}$ 。对于叶子节点，系统需要读入和其相关的所有文件块，开销为  $R_{rdm} \times L_i$ 。

#### 算法 3-1 FBX-Tree 节点重建算法

算法的输入：节点号 N。

算法的输出：节点 Nd。

01.If Nd 在缓冲区中

02. 返回 Nd;

03.Else if Nd 是一个内节点

04. 得到 Nd 的块地址;

05. 读取文件块到 Nd;

06. 返回 Nd;

07.Else \ Nd 是一个叶子节点

08. 得到 Nd 的文件块指针链表;

09. For 链表中的每一个元素

10. 读取文件块地址;

11. 抽取与 Nd 相关的信息;

12. 返回 Nd;

FBX-Tree 上的搜索操作在结构中搜索给定键值的记录。操作执行时首先检查记录是否在日志缓冲区中，若不在，用标准 B-Tree 搜索方法搜索逻辑 B-Tree。此时需要遍历从树根到叶子的节点，故搜索操作的代价为重建内节点和

重建叶子节点两部分相加，为  $R_{rdm} \times (H-1) + R_{rdm} \times L_i = R_{rdm} \times (H-1 + L_i)$ 。

### 算法 3-2 FBX-Tree 搜索算法

算法的输入：键值 K。

算法的输出：搜索结果，返回 NULL 如果没有找到具有键值 K 的记录。

01. 在缓冲区中检查是否有相关记录；
02. If 找到记录
03. 返回记录；
04. Else //搜索逻辑 B-Tree
05. 从根节点到叶子节点
06. 重建路径上的每个节点；
07. 使用 B-Tree 搜索策略搜索该节点；
08. If 找到记录
09. 返回记录；
10. 返回 NULL；

FBX-Tree 的插入操作向结构中插入一条记录。首先用 B-Tree 搜索算法定位欲插入的叶子节点，然后生成对应的日志项，插入日志缓冲池中。在这个过程中如果遇到节点已满，需要适时的分裂节点。插入操作的代价为系统读入由根到叶子的父亲节点的所有内节点的开销，加上日志缓冲区中内容写到文件后的平摊开销，综合起来为  $R_{rdm} \times (H-1) + W_{seq} / D$ ，远小于 BFTL 开销。

### 算法 3-3 FBX-Tree 插入算法

算法的输入：记录 R。

算法的输出：插入结果（成功或者失败）。

01. 从根节点到叶子节点的父亲节点
02. 重建路径上的每个节点；
03. 使用 R 的键值搜索节点；
04. 得到叶子节点号；
05. 为记录 R 准备一个日志项；
06. 将该日志项插入缓冲区；
07. If 缓冲区已满
08. 将缓冲区文件块追加到数据文件；
09. 更新相关的节点转换表入口；

FBX-Tree 的删除操作的执行过程与插入操作类似，功能为删除系统中指定键值的记录。操作执行时首先搜索日志缓冲区，若记录不在日志缓冲区中，

用该键值搜索逻辑 B-Tree，若可以找到具有该键值的记录，根据键值和该叶子节点生成删除日志，插入日志缓冲区中。若没有找到，对系统不做任何修改。本操作在最坏情况下需要对 FBX-Tree 进行搜索操作后进行日志插入操作，故其代价为  $R_{rdm} \times (H-1+L_i) + W_{seq} / D$ 。

#### 算法 3-4 FBX-Tree 删除算法

算法的输入：键值 K。

算法的输出：删除结果（True 或者 False）。

01. 检查记录是否在缓冲区中；
02. If 找到记录
03. 删除该记录；
04. 返回 True；
05. Else
06. 从根节点到叶子节点
07. 重建路径上的每个节点；
08. 使用键值 K 搜索节点；
09. If 找到拥有键值 K 的记录 R
10. 使用 K 和 R 构造删除日志；
11. 插入删除日志到缓冲区；
12. If 缓冲区已满
13. 将缓冲区文件块追加到数据文件；
14. 更新相关的节点转换表入口；
15. 返回 True；
16. Else
17. 返回 False；

在节点满时，需要对其进行分裂操作，分裂后的两个节点均指向原节点的父亲节点。对于 FBX-Tree，当节点分裂时，需要首先对待分裂节点和其父节点进行重建操作，将它们读入内存，然后分裂节点，并更新父节点指针。因为节点分裂操作通常伴随着插入操作，所以此时得到的三个节点（父亲节点和两个儿子节点）均要插入节点缓冲区。节点分裂操作的最高代价为重建父子两个节点的代价，若儿子节点为内节点，总代价为  $2R_{rdm}$ ；若儿子节点为叶子节点，总代价为  $R_{rdm} \times (1+L_i)$ 。

#### 算法 3-5 FBX-Tree 节点分裂算法

算法的输入：父亲节点，待分裂的儿子节点。

算法的输出：分裂结果（成功或者失败）。

- 01.重建父亲节点和儿子节点；
- 02.将儿子节点分裂为儿子 1 和儿子 2；
- 03.更新父亲节点中的相关信息；
- 04.将父亲节点、儿子 1 和儿子 2 插入节点缓冲区；

### 3.3 文件块指针链表维护策略

由 3.1 节可知，FBX-Tree 中的叶子节点信息或处于节点缓冲区中（当叶子节点分裂后），或为日志形式存放在文件中，且通常处于文件中。重建叶子节点时，需要读入节点转换表中与该叶子对应的文件块指针链表指向的所有文件块。因此，文件块指针链表长度决定了系统的查询性能。为此，我们定义了文件块指针链表整理操作（见 3.3.1 节），在链表过长时对其进行整理，同时将合并后的节点信息写回文件。由于该操作会使文件大小增加，且有一定开销，所以不能频繁触发，故存在一个对整理操作时机进行决策的问题。

#### 3.3.1 问题的提出

对 FBX-Tree 定义文件块指针链表整理操作，该操作首先重建叶子节点，然后将节点转换表中节点相应的链表清空，将整理后叶子节点写回文件，更新文件块指针链表信息。整理操作算法如下：

算法 3-6 FBX-Tree 链表整理算法

算法的输入：节点号 N。

算法的输出：整理结果。

- 1.重建节点 N；\N 是一个叶子节点
- 2.清空 N 的节点转换表链表；
- 3.将节点 N 追加至数据文件末端；
- 4.插入文件块号至 N 的链表中；

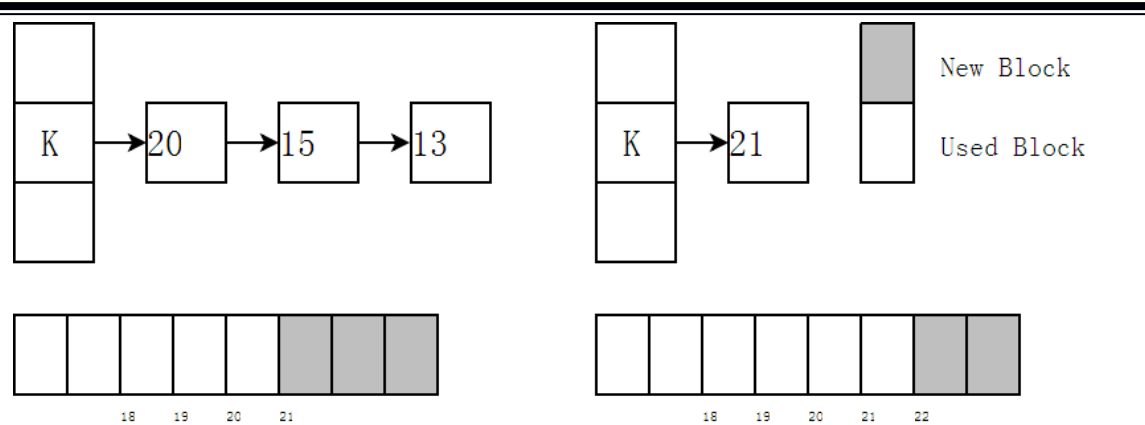


图 3-2 日志整理操作示意图

Figure3- 2 Example of Log Clean Operation

图 3-2 给出了一个日志链整理操作的例子，整理前，节点 K 对应文件块 20、15、13，从 21 开始为新文件块。整理后，文件块 20、15、13 中相应的部分失效，所有内容写到文件块 21 中，NTT 中节点 K 的日志链清空，然后添加指针指向了文件块 21，文件空闲块指针后移到 22。

对于一个将信息存放在文件块中的叶子节点，我们对其定义 3 种操作：读操作，写操作和整理操作。读操作即为 3.2 节中所述的重建节点操作。写操作将日志写入日志缓冲区，并在某时刻依照缓冲区替换策略替换到 Flash 存储器上。整理操作前面已经叙述。设对于 Flash 存储器， $R_{seq}=R_{rdm}=W_{seq}$ ，并设其为单位 1，那么读操作开销为文件块指针链表的长度  $L_i$ ；写操作开销被写日志页均摊，为  $1/D$ ，其中  $D$  为日志页可容纳日志数；整理操作开销为读操作开销加写回开销，即  $L_i+1$ 。

举例：我们用  $w$  代表写操作， $r$  代表读操作， $c$  代表整理操作。对于某叶子节点有如下操作序列  $(w, r, w, r, r, w, r, w, w, r, r, w, r)$ ，并允许系统进行一次整理操作  $c$ 。图 3-3 显示了该操作序列。其纵轴为按时间增长的操作序列，横轴显示每个操作的开销，一个格子代表一个单位的开销。由于写开销在各种情况下不变，且均摊后较少，故不计入代价计算中。若不进行整理操作，整个操作序列的开销为 24；若系统在第 2 次读操作后进行整理，将使以后每次读前两个文件块时的操作时的开销都转化为读一个文件块的开销，总开销可以节省  $5-1=4$  个单位（黑色部分）

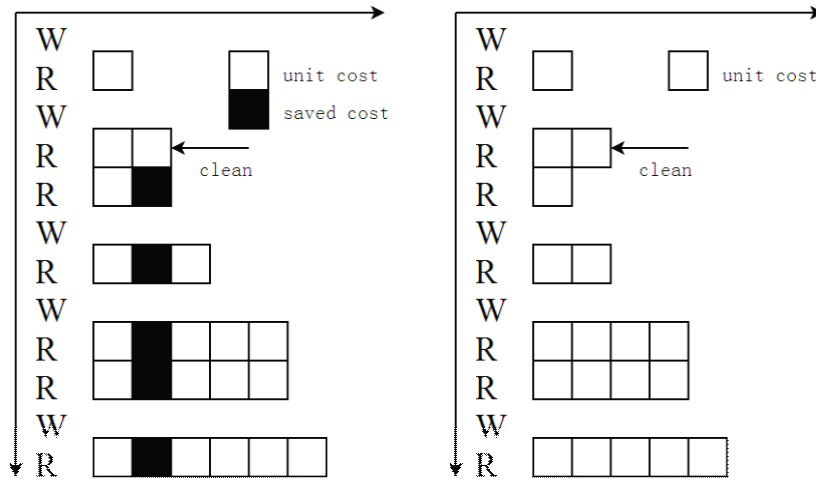


图 3-3 日志整理决策示例 1

Figure3-3 Example 1 of Log Clean Decision

若整理操作发生在第 4 次读操作之后，那么同理可以节省 5 个单位的开销，如图 3-4。

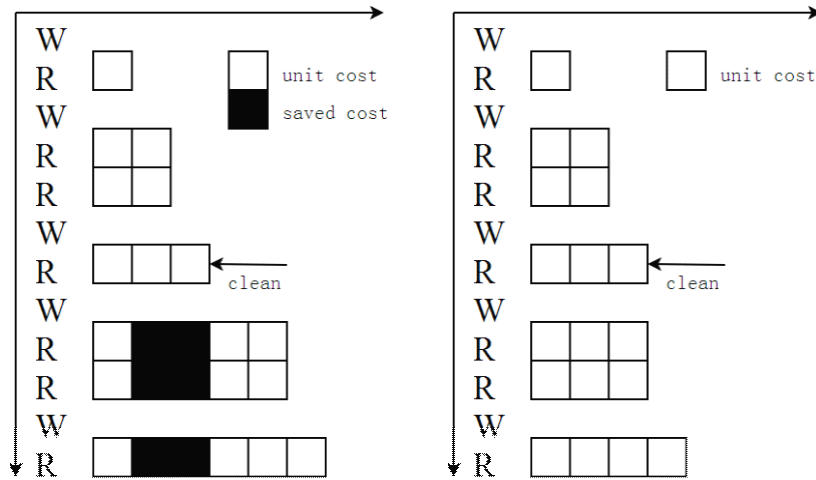


图 3-4 日志整理决策示例 2

Figure3-4 Example 2 of Log Clean Decision

由此可见对于一个给定的读写操作序列，在不同的位置触发整理操作时，可以节省的开销也不同。图 3-5 分别计算了对于例子中的操作序列在各个读操作后进行整理可以节省的开销，写在每行的最后。可见对于此操作序列，在第 5 个读操作后进行整理可以达到最小开销，节省  $8-1=7$  个单位（画\*的位置）。

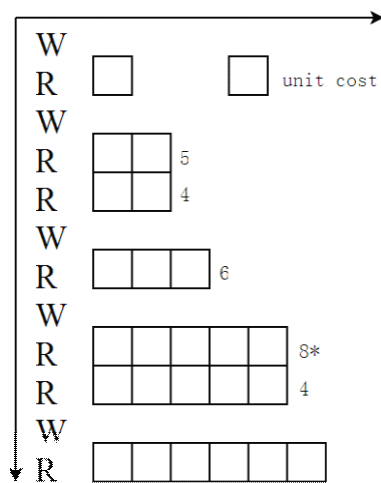


图 3-5 日志整理位置的计算

Figure3-5 Calculation for the Log Clean Position

对于每一个给定的读写操作序列，至多可以在每次读操作后进行整理操作。将各种整理操作位置穷举，我们总可以给出一个使用最少次整理操作后节省最多开销的操作序列组合。但由于读写操作序列不可知，我们无法计算出这个最优解，因此这个问题需要用 on-line 算法来解决。

问题定义：给定读写操作序列  $O=(O_1, O_2, \dots, O_n)$ ，其中  $O_i \in \{r, w\}$ ，和一个整理操作集合  $C$ ， $|C|=m$ 。将  $C$  与  $O$  结合后形成新的操作序列  $P=(P_1, P_2, \dots, P_{m+n})$ ，其中  $P_i \in \{w, r, c\}$ ，且该序列中  $r, w$  的相对位置与  $O$  中相同。用  $Cost(i)$  表示序列  $P$  中  $P_i$  项的开销，输出  $\sum_{i=1}^{m+n} Cost(i)$  开销最小时的序列  $P$ 。

当  $P$  为最优解时，关于整理操作有以下性质：

性质 3-1 最优整理操作必伴随着读操作结束进行。

若整理操作不发生在读操作结束时，那么其必然发生在写操作结束时或读操作进行中。若在写操作结束时进行整理，可以将其与该写操作后最近的读操作合并，得到的总开销更少；若在读操作进行中整理，可以将其延迟到读操作完成后整理，得到的开销更少。

性质 3-2 最优整理操作所在的读操作必然在一个写操作之后进行。

若最优整理操作所在的读操作之前不是写操作，那么其之前必然至少有一个读操作，将整理操作移到此读操作，可以得到一个更优的解。

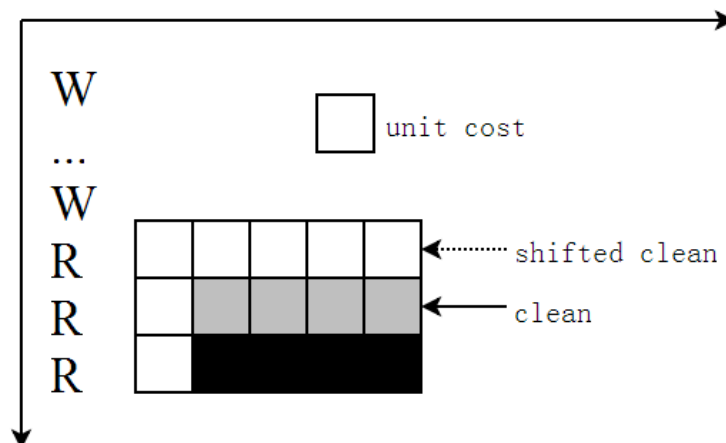


图 3-6 性质 3-2 举例

Figure3-6 Example of Property 3-2

如图 3-6 所示，将整理操作提前，可以节省更多的读开销。

性质 3-3 若读写操作序列由  $i$  个读操作和  $j$  个写操作组成，其读开销下界为  $i+j-1$ 。

对于每一个新的写，系统至少读其一次，开销为  $j$ 。对每一个读操作至少有开销  $i$ 。第一个写有可能计算两次，所以总的开销下界为  $i+j-1$ 。

### 3.3.2 三种解决方法

针对 3.3.1 节提出的问题，本节提出 3 种算法予以解决。其中算法 3-7 是以往工作中普遍使用的方法，算法 3-8 和算法 3-9 是本文提出的算法。分析与实验结果表明算法 3-9 在制造较小的数据文件同时保持了最短的链表长度。

算法 3-7 设定阈值  $K$ ，对每叶子个节点设置读操作计数器，对节点进行一次读操作后自加一。当计数器超过  $K$  时则整理本节点日志链。这也是之前的文献中普遍采用的方法。

算法 3-7 简单链表整理算法

算法的输入：阈值  $K$ ，操作序列。

算法的输出：进行整理操作的时间点。

1. acc = 0;
2. switch(operation)
3. case write:
4. 将记录写入缓冲区;
5. case read:
6. acc++;

7. if(acc > K)
8. 整理链表;
9. acc=0;

例：设 K 为 6，且有操作序列如下，则每次读操作后计数器加 1，图 3-7 中的整理操作将发生在第 6 次读操作以后。

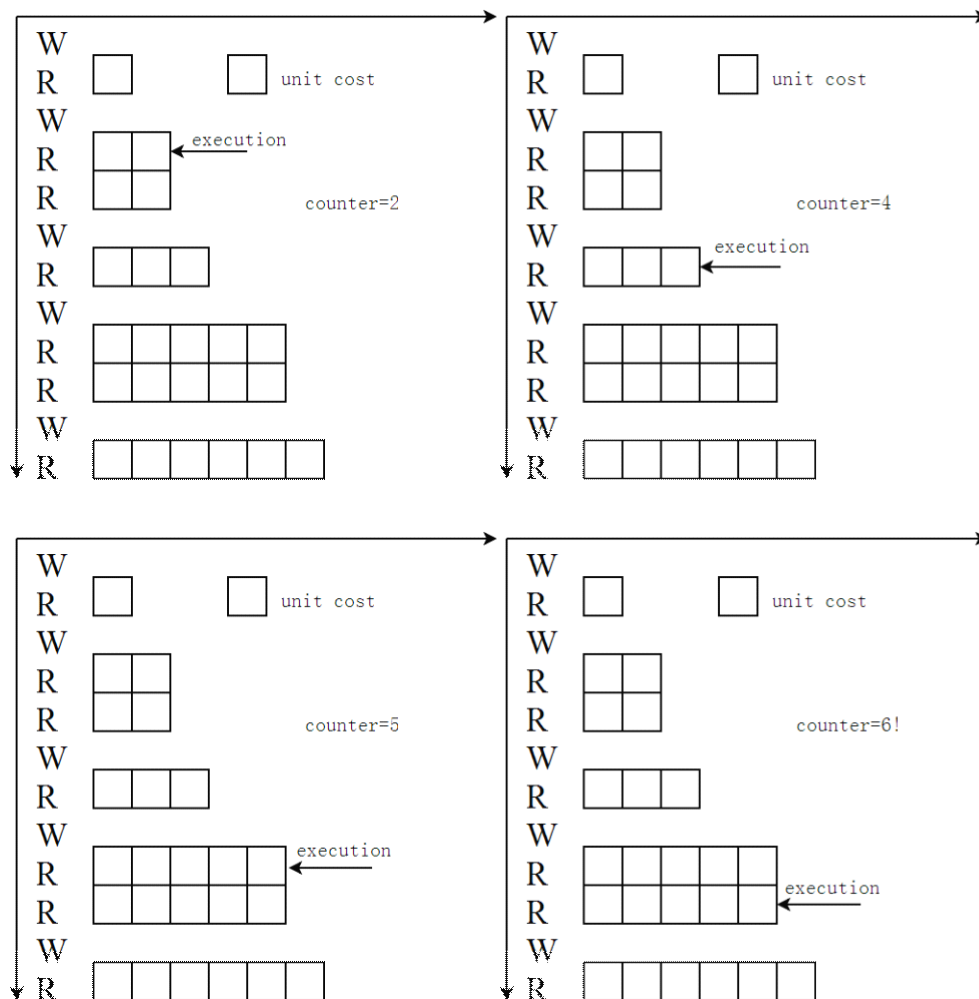


图 3-7 算法 3-7 举例

Figure3-7 Example of Algorithm 3-7

性质 3-4 设有读写序列  $O=(O_1, O_2, \dots, O_{m+K})$ ，其中有写操作  $m$  个，读操作  $K$  个，则序列在形如  $O_i=w (0 < i \leq m)$ ， $O_j=r (m+1 \leq j \leq m+K)$  时代价达到最大。

一般情况下，读写序列  $O$  的操作代价为  $\sum_{i=1}^K R(i)+1$ ，其中  $R(i)$  表示第  $i$  个读操作的代价。由于系统只在第  $K$  次读时整理，所以  $R(i) \geq R(i-1)$ ，当序列如性质 3-4 假设时，所有  $R(i)$  都等于  $R(K)$ ，所以代价取得最大值  $KR(K)+1$ 。

定理 3-1 算法 3-7 的近似比为  $K$  ( $K$  为给定阈值)。

证明：按照性质 3-4 构造读写序列  $O$ ，使算法 3-7 达到开销上界，最优操作到达开销下界，如此，算法 3-7 的操作代价为  $KR(K)+1$ ，最优代价为  $R(K)+K-1$ ，得到近似比为  $\frac{KR(K)+1}{R(K)+K-1}$ ，化简得  $K$ 。

可见，该算法的近似比较大，且随阈值的设置不同线性增长。

算法 3-8 本算法的思想与 Ski-Rental 问题的解法类似。对每个叶子节点设置计数器，计数器在每次读操作后增加该节点日志链长度大小，当当前计数器大于当前日志链长度后，日志整理操作触发。

算法 3-8 Longer Than Next 算法

算法的输入：操作序列。

算法的输出：进行整理操作的时间点。

```

01. acc = 0;
02. switch(operation)
03.   case write:
04.     将记录写入缓冲区;
05.   case read:
06.     if(acc > log length)
07.       整理链表;
08.       acc=0;
09.     else
10.       acc = acc + log length;
    
```

例如，有操作序列如图 3 所示，执行算法 2 将在第 3 个读操作后触发整理操作。

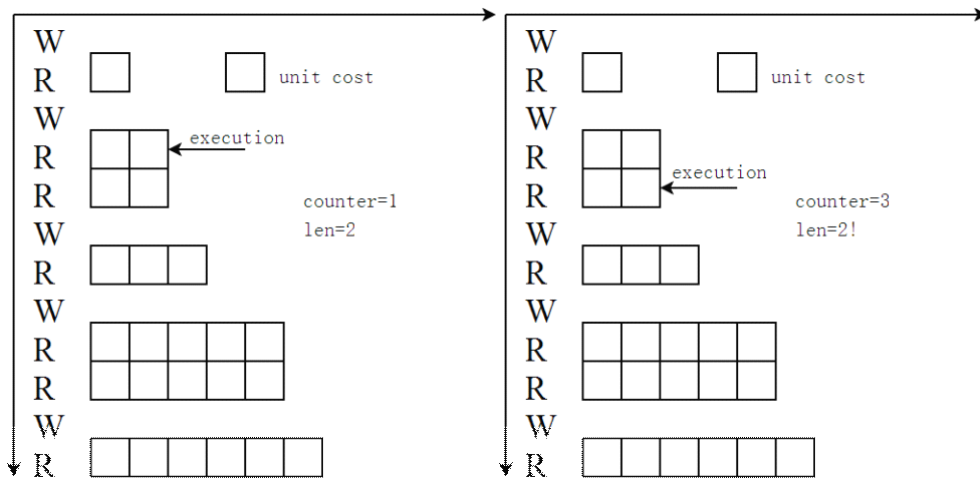


图 3-8 算法 3-8 举例

Figure3-8 Example of Algorithm 3-8

定理 3-2 算法 3-8 的近似比为 3。

证明：设一个操作序列中读操作为  $(R_1, R_2, \dots, R_n)$ ，经算法 2，在  $R_n$  处进行整理。若  $R_i$  指代在  $i$  处触发读操作产生的开销，由方法 2 的运行过程可得

$$\sum_{i=1}^{n-1} R_i > R_n, \quad \sum_{i=1}^{n-2} R_i < R_{n-1}, \quad \text{且 } R_{n-1} < R_n, \quad \text{故}$$

$$\sum_{i=1}^n R_i = \sum_{i=1}^{n-2} R_i + R_{n-1} + R_n < 2R_{n-1} + R_n < 3R_n, \quad 2R_n < \sum_{i=1}^n R_i < 3R_n. \quad \text{由性质 3-3 可知,}$$

最优开销的下界超过  $R_n$ ，所以本方法的近似比至少为 3。

本方法的近似比较小，但触发整理操作的次数过于频繁，使得数据文件增长过快，实际表现并不是特别理想。

算法 3-9 该算法的思想为算法 3-7 与算法 3-8 的结合。为每个节点设置计数器，并设定阈值  $K$ 。每次读操作后，计数器增加日志链长度大小，当计数器大于阈值  $K$  后，日志整理操作触发。

算法 3-9 Accumulate And Clean 算法

算法的输入：阈值  $K$ ，操作序列。

算法的输出：进行整理操作的时间点。

01. acc = 0;

02. switch(operation)

03. case write:

04. 将记录写入缓冲区;

05. case read:

- 06. acc = acc + log length;
- 07. if(acc > threshold)
- 08.     整理链表;
- 09.     acc=0;

若有操作序列如图 3-9 所示，执行算法 3-9，设阈值为 10，则执行到第 5 个读操作时触发整理操作。

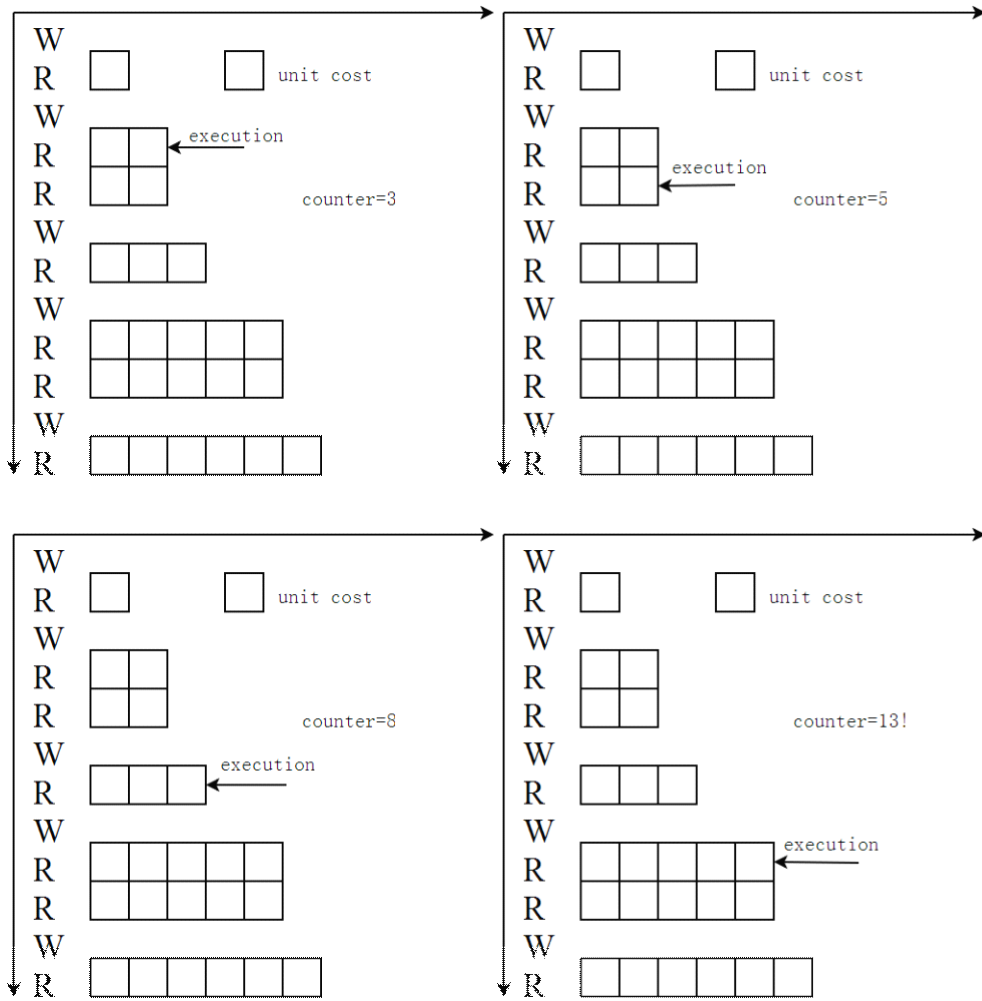


图 3-9 算法 3-9 操作示例

Figure 3-9 Example of Algorithm 3-9

定理 3-3 算法 3-9 的近似比为  $\frac{\sqrt{K}}{2} + \frac{1}{3}$ 。

证明：设使用算法 3-9 时进行了 N 次整理，则总开销为 NK+N。可以构造操作序列如图 3-10，使最优操作达到性质 3-3 所描述的下界，最优操作在每一个 wr 转换的位置进行。设 w 长度为 W，则 r 长度为 K/W，则最优操作代价

为  $N(W+K/W+1)$ ，最优操作代价在  $W=\sqrt{K}$  时达到最小。这时使最优开销达到下界，近似比为  $\frac{K}{2\sqrt{K}-1} = \frac{\sqrt{K}}{2} + \frac{\sqrt{K}}{4\sqrt{K}-2} < \frac{\sqrt{K}}{2} + \frac{1}{3}$ 。

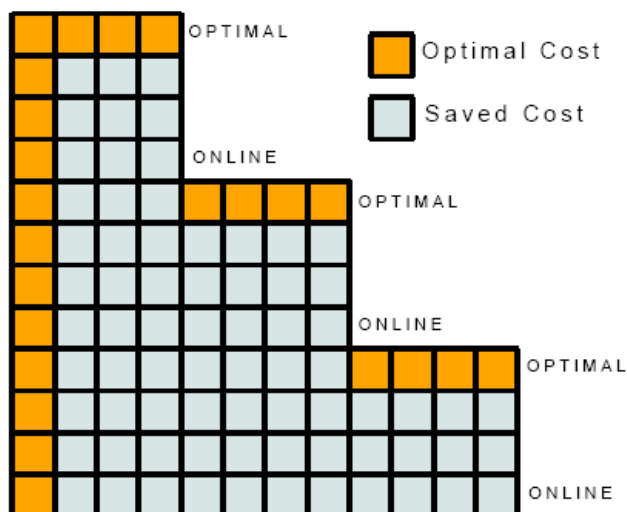


图 3-10 构造操作序列

Figure3-10 Operation Sequence Construction

### 3.4 本章小结

本章介绍了 FBX-Tree 的构造方法及维护策略包括 FBX-Tree 的插入、删除、查询操作，然后提出了文件块指针链表的整理问题并将其形式化，并提出了三个 on-line 算法解决该问题。我们证明了这三个算法的近似比分别为  $K$ 、 $3$  和  $\frac{\sqrt{K}}{2} + \frac{1}{3}$ （其中  $K$  为算法给定值），其中算法 3-9 在制造较小的数据文件同时保持了最短的链表长度，从而确保了高效的查询性能。

## 第 4 章 基于 FBX-Tree 的移动对象查询处理方法

为展示 FBX-Tree 的使用方法和性能，本章使用 FBX-Tree 解决移动对象的查询处理问题。本章设计了一个移动对象数据库原型系统，该系统后端存储部分使用 FBX-Tree 实现，前端索引部分使用与 B<sup>x</sup>-Tree 相似的策略，将对象空间进行降维后存储到逻辑 B-Tree 中，并在查询处理部分对 FBX-Tree 做了优化。

### 4.1 移动对象数据库原型系统设计

本章设计的原型系统主要由查询处理模块、存储管理模块、缓冲区管理模块几部分组成。其中查询处理模块主要负责将用户查询转换为一维查询，分配到相关的逻辑 B-Tree 上执行。存储管理模块负责管理逻辑 B-Tree 的内存和外存之间的协调工作，缓冲区管理模块负责读写缓冲区和日志缓冲区的管理。整体框架如图 4-1 所示。

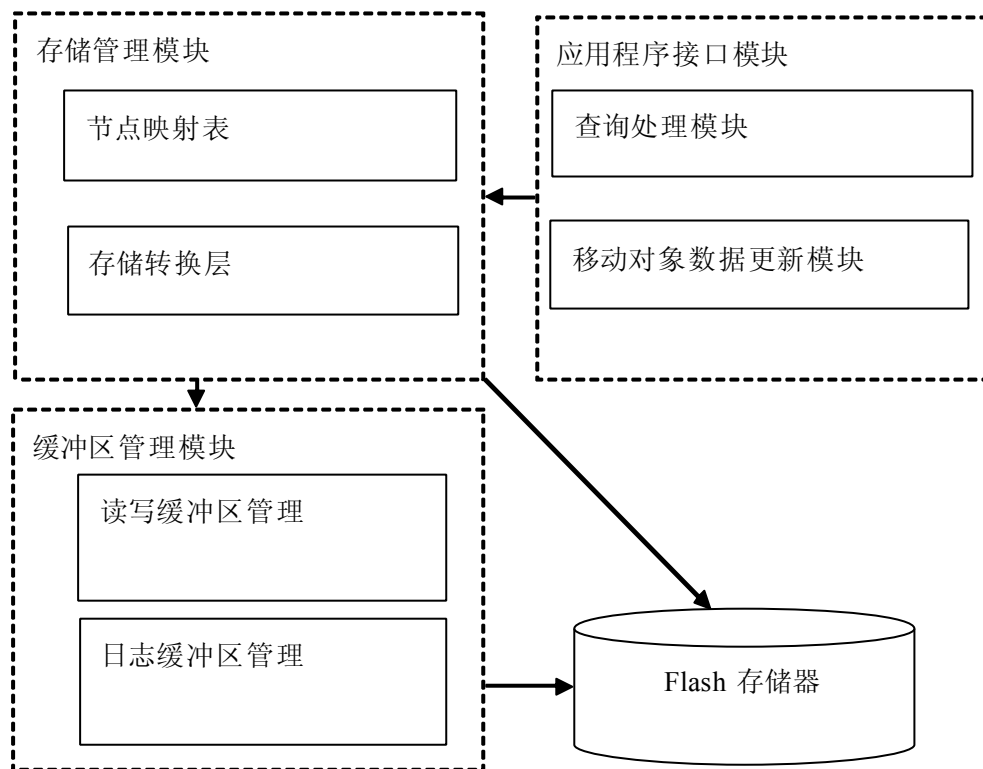


图 4-1 原型系统框架

Figure4-1 Framework of the Prototype

我们使用 FBX-Tree 作为存储后端索引移动对象，使用 B<sup>x</sup>-Tree 的方法将二维数据映射到逻辑 B-Tree 上。由于在其上每个时段对应一棵逻辑 B-Tree，所以对每一棵逻辑 B-Tree 建立一个 FBX-Tree 实例，在实例工作期间不创建新的数据文件，各个文件轮转工作，每个文件都不会过大。

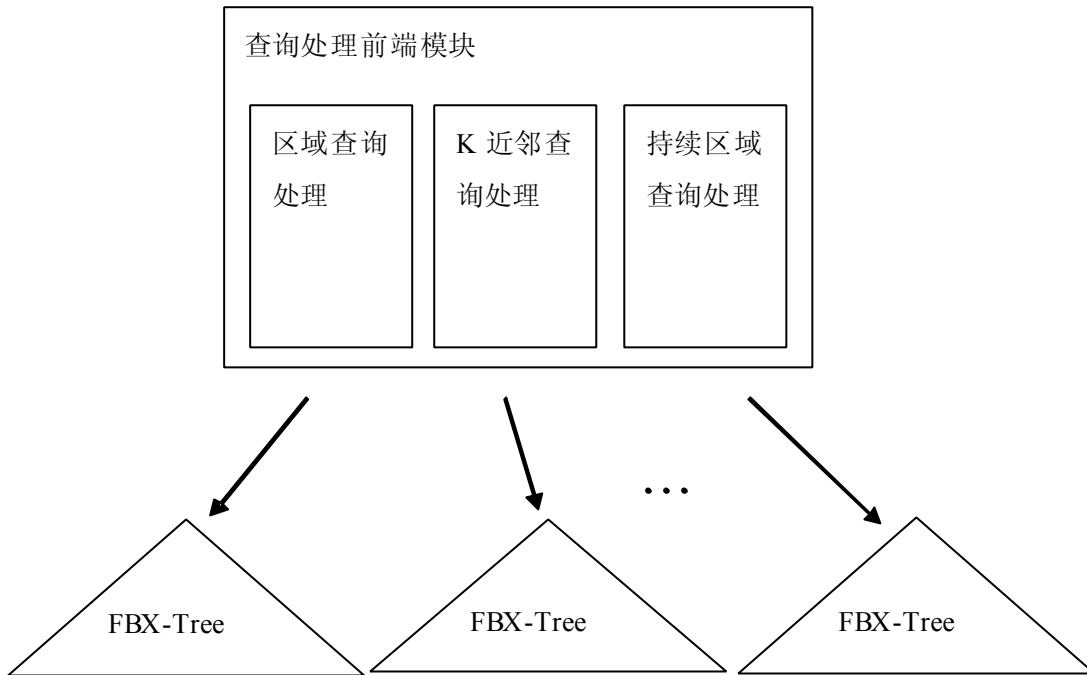


图 4-2 移动对象索引结构示意图

Figure4-2 Example of Moving Object Index

正常情况下，每棵 FBX-Tree 的叶节点日志数据分散到对应数据文件的文件块中，需要相关节点时，需要读入这些文件块。若可以为这些日志数据保留一定的对象聚集特性，那么相关的文件块指针链表长度将缩短，查询性能也将得到提高。

我们将日志缓冲区的大小设定为 2 的指数倍，并为其编号。当数据到来时，将其所在的网格编号前几位与缓冲区号进行匹配，放入相应的缓冲区中。以保留聚集特性。在本方案中，我们设置 1MB 的日志缓冲区，如此网格编号的前 8 位都可以用来进行缓冲区划分。

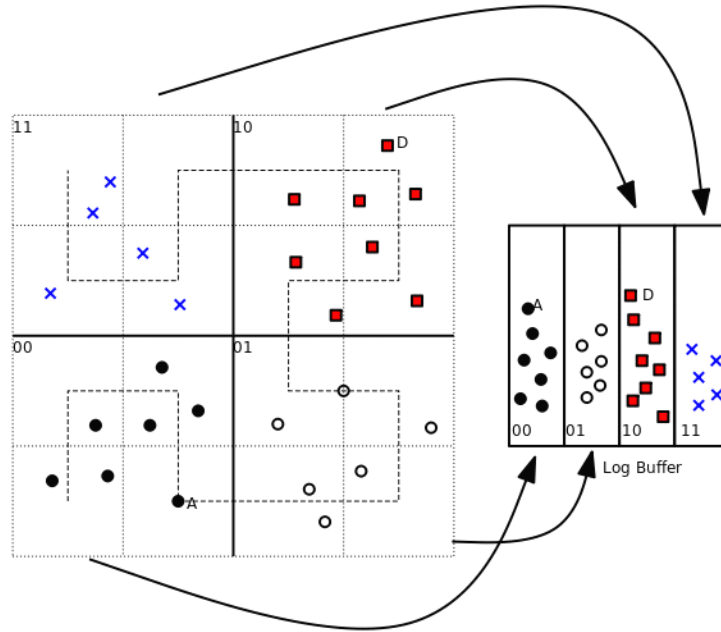


图 4-3 日志缓冲区的聚集特性举例

Figure4-3 Example for the Log Buffer Cluster Property

如图 4-3 所示，系统设置 4 个页大小的日志缓冲区。当数据到来时，根据其所在网格二进制编号前缀的不同进入不同的缓冲区中，每个缓冲区充满后写到文件末端。如本例中的点 A 的信息存放于缓冲区 00 中。

## 4.2 区域查询 (Rang Query)

区域查询定义如下：给定查询  $q=(r,[t_1,t_2])$ ，其中  $r$  为查询区域， $[t_1,t_2]$  为查询对应的时间区间，系统返回在  $[t_1,t_2]$  时刻内经过区域  $r$  的所有对象。

与  $B^x$ -Tree 类似，FBX-Tree 将区域查询转换为 B-Tree 上的一系列一维区域查询执行。通过判断哪些曲线经过查询区域，系统可以得到一系列的一维区域片段，使用这些片段进行区域查询，就可以得到结果。在执行区域查询时，由于日志保留了一定的聚集特性，若依次重建每个叶子节点，将可能重复读取某一文件块数次，故需要在读取页之前合并相同的文件块，接着读取每个文件块一次得到结果。FBX-Tree 的区域查询的算法如下：

### 算法 4-1 区域查询算法

算法的输入：Range  $r$ , Time-interval  $[t_1,t_2]$ .

算法的输出：Objects in the range  $r$  with the time stamp between  $t_1$  and  $t_2$ .

01.转换  $Q$  到相关的 FBX-Tree 上；

02.For 每一棵 FBX-Tree

03. 建立地址集合 A;
04. 建立节点集合 N;
05. 将相关节点放到 N 中;
06. For 每个 N 中的节点
07.     从节点转换表得到链表;
08.     For 链表中的每个地址
09.         插入地址到 A;
10.     For A 中的每个地址
11.         读取地址指向的文件块;
12.         For 文件块中的每个日志项
13.             If 日志属于 N 中的节点
14.                 更新该节点信息;

进行区域查询的代价与区域大小及区域内点的密度有关。若查询 Q 可转化为 N 个一维区域查询，并需要读取叶节点 M 次，则该查询的代价为  $N \times (H-1) \times R + M \times L$ 。

### 4.3 K 近邻查询 (K Nearest Neighbor Query)

K 近邻查询定义如下：给定查询点 p 和阈值 K，返回当前时刻在对象空间中距离点 p 最近的 K 个对象。

执行 K 近邻查询时，系统首先以查询点为中心做一个网格大小的区域查询，将其放到 FBX-Tree 上执行。若没有返回足够的结果或者结果可能包含在其他网格中，则扩展现有区域查询继续返回结果直至数量达到 K。

与区域查询类似，在进行 K 近邻查询时，系统需要重建相邻叶子节点，查询点附近的节点。有可能重新读取某些磁盘块数次。故在进行页读取之前需要合并相同的磁盘块。扫描每个磁盘块一次得到最终结果。

算法 4-2 KNN 查询算法 (Query Point p, Threshold K)

算法的输入：Query Point p, Threshold K。

算法的输出：The K nearest neighbour of p。

01. R=∅;
02. 在树中搜索 p，在节点 N 中找到;
03. 将节点 N 中的点放到 R 中;
04. While |R|<K
05.     扩大查询区域;

06. 将新的点加入 R 中;

07.返回 R;

#### 4.4 持续区域查询 (Continual Range Query)

定义：给定区域查询集合  $Q=\{q_1, q_2, q_3, \dots, q_n\}$ ，其中  $q_i$  为传统的区域查询，系统持续的返回符合  $q_i$  的结果集。

由于系统需要持续返回结果，若使用常规的区域查询方法对其进行处理，将结果存储于外存中，则需要周期性的执行大量区域查询语句，其无法满足结果的实时更新和吞吐率需求，故多数处理持续区域查询的方法将查询结果置于内存中。

文献[37]的处理方法为在内存中为每个查询保留结果集，在数据更新时更新结果集。若查询区域有重叠，在数据更新的时候，相关查询的结果集都需要更新，并且需要占用额外的宝贵的内存空间。基于查询集合不频繁变化的特点，文献[45]等提出系统为查询建立索引。首先将查询合并、分解为一系列不相交的子查询，组织成网格结构存储在内存中。

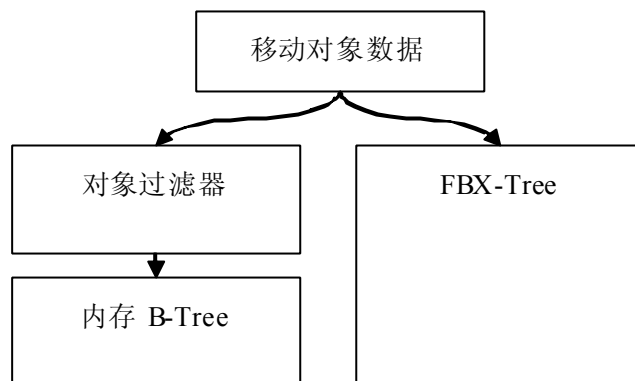


图 4-4 处理持续区域查询方案结构图

Figure4-4 Framework of the Continual Range Query Processing

本方案中使用了类似的策略，并根据 FBX-Tree 自身的特点，对移动对象进行索引，内存中只保留结果集中对象的一个副本，减小了更新代价，并减少了内存占用。

在本方案中，系统在内存中维护一个过滤器 F 和一棵 B-Tree。移动对象信息通过过滤器过滤后被索引在内存 B-Tree 中。持续查询集合中的查询指向 B-Tree 叶子节点，可以持续的返回查询结果。

下面给出处理持续区域查询的具体示例。如图 4-5 所示，空间被划分为  $16 \times 16$  的网格，并用 Hilbert 曲线填充。给定系统的持续查询集合  $Q=\{Q1, Q2,$

Q3, Q4}。覆盖查询区域用灰色表示并存在重叠。用 Hilbert 曲线降维后，这些查询对应了以下一维区域查询集合

Q1:(L11,R11),(L12,R12)

Q2:(L21,R21),(L22,R22),(L23,R23),(L24,R24),(L25,R25)

Q3:(L31,R31)

Q4:(L41,R41),(L42,R42),(L43,R43),(L44,R44)

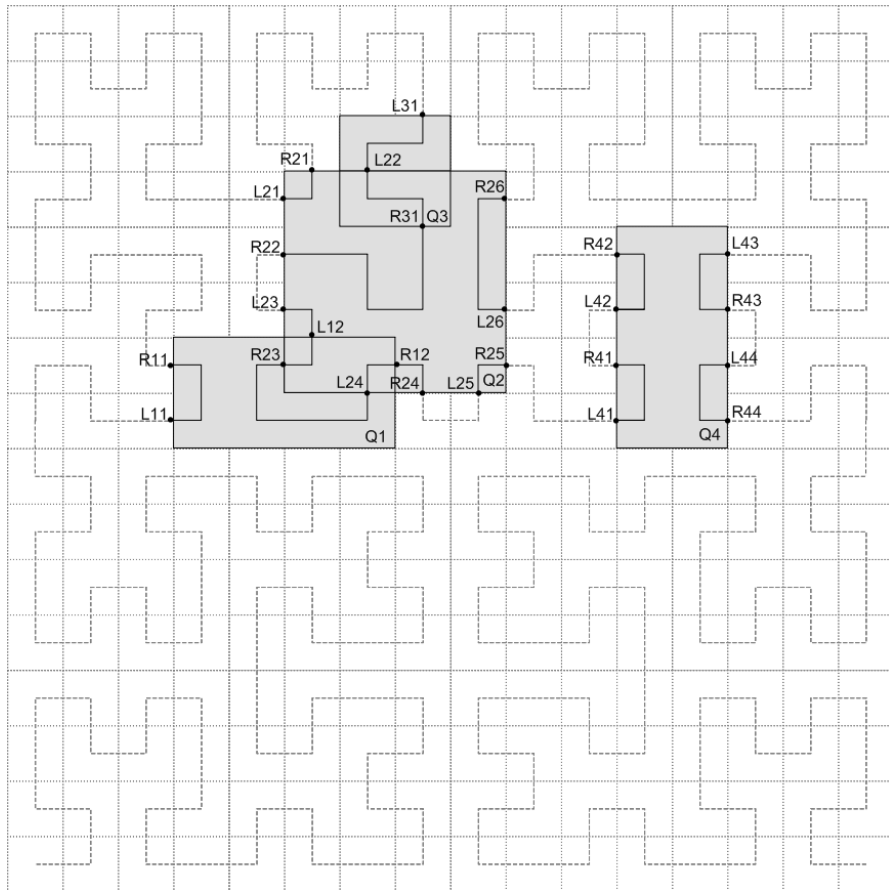


图 4-5 持续区域查询举例

Figure4-5 Example of Continual Range Query

对重叠区域进行合并后得到不相交的区间：(L11,R11), (L21,R21), (L31,R22), (L23,R24), (L25,R25), (L26,R26), (L41,R41), (L42,R42), (L43,R43), (L44,R44)。将其作为过滤器区间，点在这些区间中时则将其放在内存中。图中对象 O1 在区域(L23,R24)中，所以被索引。图中对象 O2 不在任何区域中，所以被过滤。

过滤器的构造是实现查询处理的重要步骤。构造过滤器算法的输入为一个持续查询集合，最终构造结果为一系列不相交的区域序列。算法首先将每个区域查询分解为多个一维查询，然后将这些一维查询合并为多个不相交的一维区

间。将区间按升序排列，当新点进到系统中时，对此排列使用二分查找即可。

过滤器的构造算法如下：

算法 4-3 过滤器构造算法

算法的输入：Query Set Q。

算法的输出：Filter F。

- 01.将 Q 转换成一维查询集合 Q'；
- 02.将 Q'合并为不相交查询集合 Q''；
- 03.由 Q''构造过滤器 F；
- 04.返回 F；

例如有过滤器 (1.3L, 2.8R, 3.3L, 7R, 9.5L, 11R)，则在其中搜索点 9.8 时搜索到位置 9.5L 与 11R 之间，说明改点在索引范围内。

在过滤器构造过程中需要对一维区域进行合并，合并结果为不相交的区域。合并方式如下：将区域的左右区间分别看成点，用快速排序排列成一个序列。由头到尾扫描已排序后的序列，遇到左区间，将其入栈，遇右区间则弹栈，弹栈后若栈空，则制造一个区域。

合并算法如下：

算法 4-4 区域合并算法

算法的输入：一维区域集合 R。

算法的输出：不相交的一维区域集合 D。

- 01.由 R 得到点对；
- 02.将点对升序排列；
- 03.建立栈 S；
- 04.For 每一个点序列中的点 k
05. If k 是某个区域的左端点
06. 将 k 压入 S；
07. Else
08. 从栈中弹出一个元素；
09. If S 为空
10. 建立区间并加入集合 D；

例：假设有两个查询 Q1, Q2；在经过降维后，Q1 被翻译为 (1.5, 4), (5, 10), (13, 15), (22, 26)；Q2 为 (3, 5.5), (12, 13.5), (25, 27), (33, 36)；

将其左右区间看成数据点按升序排列后为 (1.5L, 3L, 4R, 5L, 5.5R, 10R,



象数据库原型系统。该系统使用  $B^x$ -Tree 的降维策略和查询处理思路，并支持区域查询、K 近邻查询和持续区域查询。我们分别对每种查询给出了详细的查询处理方法描述，并针对 FBX-Tree 的特点进行了优化。

## 第 5 章 实验结果与分析

本章对 FBX-Tree 进行了详细的试验，首先将 FBX-Tree 与 B-Tree 进行对比，然后针对不同的工作负载、不同的文件块大小、不同的日志管理策略分别对 FBX-Tree 做了性能测试。

### 5.1 实验与分析

本文实验使用 C++ 语言实现。实验用机器的配置为：Intel P4 2.8G 处理器，512MB 内存，Windows XP SP3 操作系统，GCC 编译器。Flash 设备为 OCZ 公司生产的 64G SSD。实验中使用的每个记录由键值（4 bytes）与数据（20 bytes）两部分组成。如无特殊说明，页面大小为 4KB，内节点度为 500，叶节点度为 100，系统的读写缓冲区为 8M，日志缓冲区为 1M。

#### 5.1.1 B-Tree 与 FBX-Tree 性能对比

我们首先测试了在 FBX-Tree 和 B-Tree 中插入和查询不同数量的条目后所用的时间，测试结果见图 5-2。首先向其中插入随机产生  $10^6$  个条目，分别为其在插入  $1 \times 10^5$ 、 $2 \times 10^5$  至  $10^6$  条时计时。可以发现，B-Tree 对于插入操作性能较差，耗费时间约为 FBX-Tree 的 20 倍左右，并在记录数量增加后有继续上升的趋势。对于搜索操作，B-Tree 性能表现较好，这归功于 Flash 存储器高性能的随机读速度。

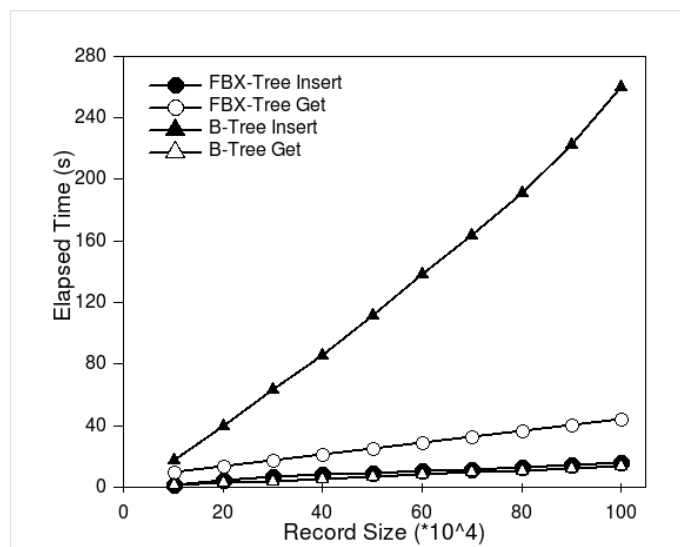


图 5-1 FBX-Tree 与 B-Tree 性能对比

Figure 5-1 Comparison Between FBX-Tree and B-Tree

### 5.1.2 工作负载对系统性能的影响

我们通过实验测试不同的工作负载环境下 FBX-Tree 的表现。为了模拟一个真实的系统使用环境，我们随机产生了 10 组实验数据（见图 5-2），其中的插入操作所占的比值由 50%到 90%，数据量分别为  $10^6$  和  $2 \times 10^6$ 。结果显示，在插入操作数量较小的情况下（50 万条以下），B-Tree 保持了平衡的读写性能，我们认为这主要归功于为其设置的 8M 缓冲区。当插入数据超过 50 万，flash 的随机写特性成为制约系统吞吐率的瓶颈，而在此情况下，FBX-Tree 对于每种工作负载都保持了较高的执行效率。

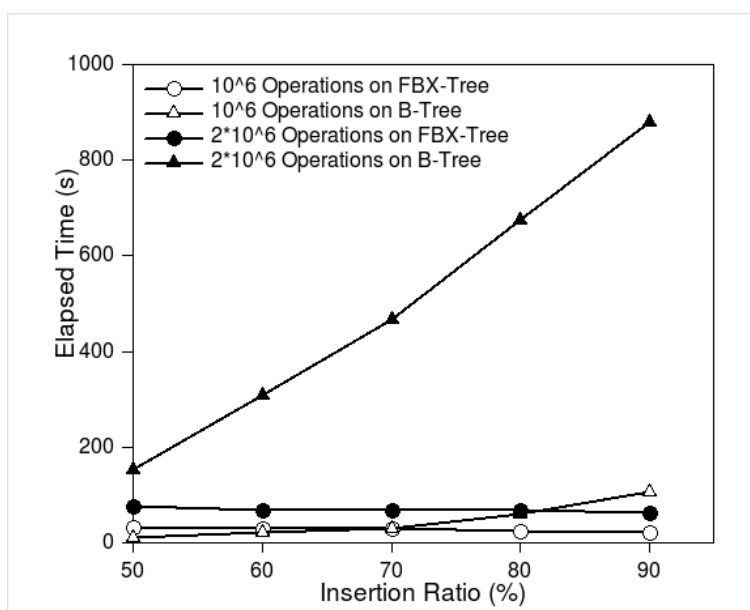


图 5-2 FBX-Tree 在各工作负载下表现

Figure5-2 FBX-Tree Performance on Different Workload

### 5.1.3 文件块大小的选取

在 FBX-Tree 中，不同的文件块大小对系统的性能有一定影响，当文件块较大时，树高较低，查询性能提高，但数据文件变大；当文件块较小时，则相反。我们分别用插入操作占 60%至 90%的负载在文件块设定为 1K、2K、4K 和 8K 的 FBX-Tree 上执行，结果显示，当文件块取为 2K 和 4K 时，系统的数据文件大小和执行时间取得了平衡。图 5-3 中显示的是当插入操作占 70%时的各 FBX-Tree 执行操作的耗时情况。

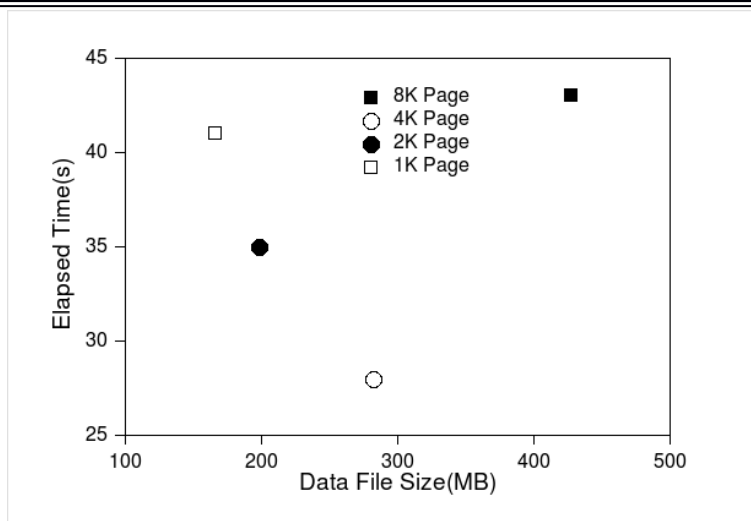


图 5-3 文件块大小的选择

Figure5-3 Selection of the Block Size

#### 5.1.4 日志整理策略的选取

本节对 3.3 节提到的三种链表管理策略进行了比较。我们在不同负载下执行  $10^6$  次操作，然后比较各策略的链表长度。我们发现算法 3-9 在各种负载下都保持了最佳效果。在写频繁的负载环境中，算法 3-8 的效果可圈可点，但是在读操作比重增加后，其性能下降剧烈。

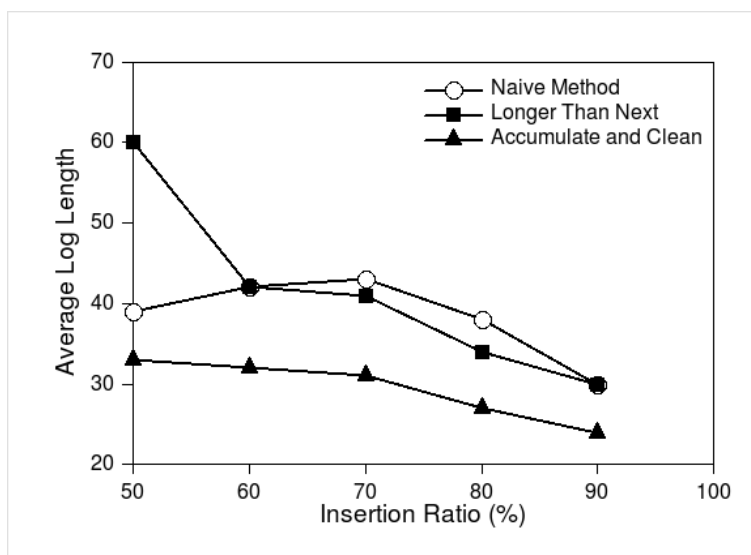


图 5-4 日志整理策略的比较

Figure5-5 Comparison of the three strategies

在算法 3-9 中，由于算法近似比与阈值大小有关，阈值的选取对系统的查询执行性能有着显著的影响。我们通过比较不同阈值、不同负载下日志的平

均长度确定了最佳阈值取值。通过图 5-5 可见，当阈值在 60 到 70 之间时，算法 3-9 对于各种负载都取得了最好的效果。

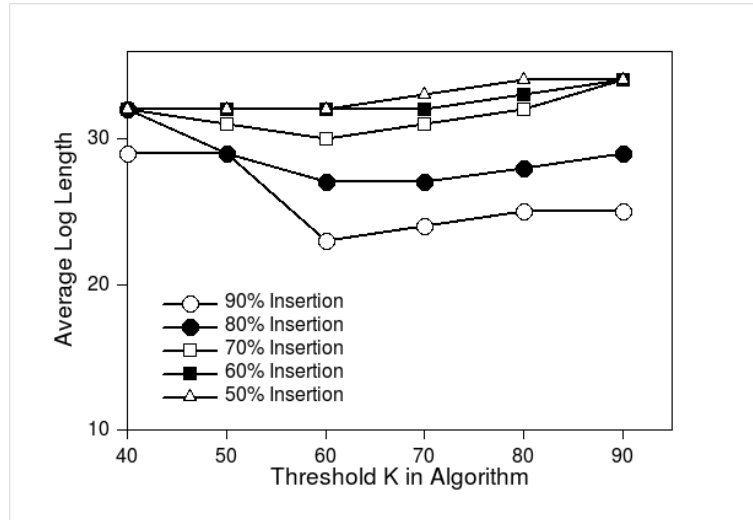


图 5-5 算法 3-9 阈值的选取

Figure5-5 Threshold Selection for Algorithm3-9

### 5.1.5 日志缓冲区大小对查询性能的影响

在 5.1 节提到，为进一步优化查询效率，我们可以通过为 FBX-Tree 设置更多的日志缓冲区，使日志信息得到聚集，保留一定的局部性特征，从而使得内存维护的日志链缩短，查询更加有效。图 5-6 显示了在设置不同大小的缓冲区后查询性能与设置 1 个日志页时的查询性能对比。结果显示，日志缓冲区大小对查询性能影响明显，当日志缓冲区设置为 256 个日志页（1MB）时，查询加速比可以接近 3 倍。

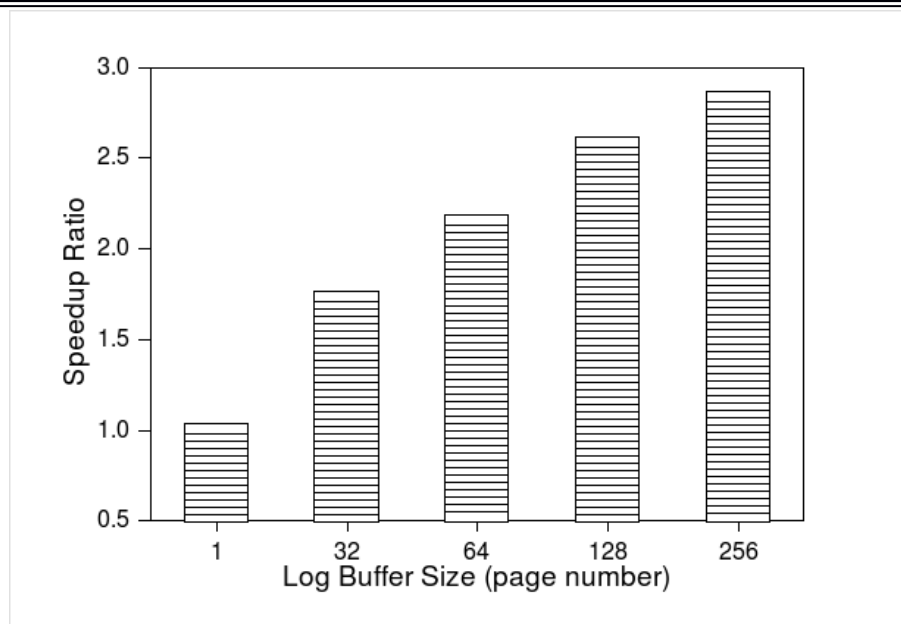


图 5-6 缓冲区大小的设定

Figure5-6 Buffer Size Configuration

## 5.2 本章小结

本章针对不同的工作负载、不同的文件块大小、不同的日志管理策略等分别对 FBX-Tree 做了性能测试。实验结果证明 FBX-Tree 在各种工作负载下表现均衡，具有良好的适应性和查询性能。

## 结论

随着 Flash 存储器的普遍使用，研究在数据频繁更新的环境下如何高效的使用 Flash 存储器变得十分必要。本文主要研究了在数据频繁更新环境下的在 Flash 存储器上进行数据管理的问题。

直接在数据频繁更新的环境中使用 flash 存储器有以下问题：首先，由于 Flash 存储器对于更新内部文件块表现较差，系统在 flash 上运行无法获得较好的性能；其次，大量的更新操作会导致 flash 器件寿命显著缩短。针对以上问题，本文提出了一种称为 FBX-Tree 的索引策略。该策略通过日志信息和缓存操作将所有的数据更新转化为对数据文件的顺序写操作，有效避免了对文件内部进行更新，保证了系统的吞吐率并延长了器件寿命。FBX-Tree 工作时，需要在内存中为每个叶子节点维护文件块指针链表。链表的长度影响了系统的查询性能。为此，本文提出了三个维护链表长度的 on-line 算法，并对它们的近似比进行了证明。这三个算法使系统的查询性能较以往有较大提高。

为验证 FBX-Tree 的工作效果，本文将其应用于移动对象数据库领域，提出一种基于 FBX-Tree 的移动对象索引方案。该方案使用 FBX-Tree 作为后端存储结构，使用 B<sup>x</sup>-Tree 作为前端查询处理前端，支持区域查询、K 近邻查询和持续区域查询。这些查询处理操作均针对 FBX-Tree 进行了重新设计和优化。

利用上述方法，本文对 FBX-Tree 进行了大量的实验。理论分析和实验结果表明，FBX-Tree 可以充分利用 Flash 存储器的读写特点，在保持查询效率的基础上，拥有更高的吞吐率，并延长了器件的使用寿命。

未来的工作分为以下几个方面。首先考虑将更多的传统磁盘环境下的写频繁策略移植到 Flash 存储器上，包括 LSM-Tree 等。其次考虑修改 Flash 存储器上的缓冲区管理策略以获得更好的页命中率。

## 参考文献

1. 曹炳乾, 张维君. 平均读写算法在闪存中的应用研究. 单片机与嵌入式系统应用. 2007 年第 9 期: 68-69 页
2. N. Agrawal, V. Prabhakaran, T. Wobber, et al. Design Tradeoffs for Ssd Performance. Proceedings of the Usenix Annual Technical Conference (USENIX' 08), Boston, MA, 2008. USENIX Association Berkeley, CA, USA, 2008: Pages 57-70
3. 王伟能, 王鹤群. 固态硬盘概述. 记录媒体技术. 2009 年 01 期: 42-46 页
4. 王伟能, 马建设, 潘龙法. 电子硬盘的 NAND 闪存地址映射策略. 记录媒体技术. 2009 年 01 期: 21-23 页
5. M. A. Shah, S. Harizopoulos, J. L. Wiener, et al. Fast Scans and Joins Using Flash Drives. Data Management On New Hardware, Vancouver, Canada, 2008. ACM New York, NY, USA, 2008: Pages 17-24
6. C. WU, L. CHANG, T. KUO. An Efficient B-tree Layer for Flash-memory Storage Systems. ACM Transactions on Embedded Computing Systems (TECS). Volume 6 , Issue 3 (July 2007) :Article No. 19
7. C.-H. Wu, L.-P. Chang, T.-W. Kuo. An Efficient R-tree Implementation Over Flashmemory Storage Systems. GIS ' 03: Proceedings of the 11th ACM international symposium on Advances in geographic information systems. New Orleans, Louisiana, USA , 2003, ACM New York, NY, USA, 2003: Pages 17 - 24
8. S. Nath, A. Kansal. Flashdb: Dynamic Self-tuning Database for Nand Flash. IPSN ' 07: Proceedings of the 6th international conference on Information processing in sensor networks, Cambridge, Massachusetts, USA , 2007. ACM New York, NY, USA, 2007: Pages 410 - 419
9. S. V. Ioannis Koltsidas. Flashing up the Storage Layer. Proceedings of the VLDB Endowment. Volume 1 , Issue 1 (August 2008): Pages 514-525
10. A. Mani, M. Rajashekhar, P. Levis. Tinx: A Tiny Index Design for Flash Memory on Wireless Sensor Devices. SenSys ' 06: Proceedings of the 4th international conference on Embedded networked sensor systems. Boulder, Colorado, USA, 2006. ACM New York, NY, USA, 2006: Pages 425 - 426

11. S.-W. Lee, B. Moon. Design of Flash-based Dbms: An In-page Logging Approach. SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. Beijing, China. 2007. New York, NY, USA, 2007: Page 55 - 66
12. D. Kang, D. Jung, J.-U. Kang, et al.  $\mu$ -tree: an ordered index structure for NAND flash memory. International Conference On Embedded Software. Salzburg, Austria. 2007. ACM New York, NY, USA, 2007: 144 - 153
13. Shaoyi Yin, Philippe Pucheral, Xiaofeng Meng. A sequential indexing scheme for flash-based embedded systems. Extending Database Technology, Vol. 360: Pages 588-599
14. Yinan Li, Bingsheng He, Qiong Luo, Ke Yi, Tree Indexing on Flash Disks. icde, 2009 IEEE International Conference on Data Engineering, 2009: pp.1303-1306
15. D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, et al. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. USENIX FAST. 2005
16. Xiang Li, Zhou Da, Xiaofeng Meng. A New Dynamic Hash Index for Flash-Based Storage. Web-Age Information Management, 2008. IEEE Computer Society Washington, DC, USA: Pages 93-98
17. S. Nath, P. B. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. Proceedings of the VLDB Endowment. Volume 1, Issue 1 (August 2008): Pages 970-983
18. Y. Diao, D. Ganesan, G. Mathur, et al. Rethinking Data Management for Storage-centric Sensor Networks. Conference on Innovative Data Systems Research. 2007: 22 - 31
19. 陈智育. 嵌入式系统中的 Flash 文件系统. 单片机与嵌入式系统应用, 2002 年第 2 期: 5-8 页
20. Bitvutskiy, A.-B. JFFS3 Design Issues. Tech. Technique Report, Nov. 2005.
21. K. Ross. Modeling the Performance of Algorithms on Flash Memory Devices. Data Management On New Hardware, Vancouver, Canada, 2008. ACM New York, NY, USA, 2008: Pages 11-16
22. 董明, 刘加. 适宜于嵌入式多媒体应用的 Flash 文件系统. 电子技术应用, 2002 年第 28 卷 第 09 期: 24-27 页
23. Deepak Ajwani et all. Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design. Lecture Notes in Computer

- 
- Science. Volume 5038/2008. 208-219
24. Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage, Proceedings of the 6th USENIX Conference on File and Storage Technologies. San Jose, California. Article No. 16
  25. Shimin Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In Proceedings of the 28th ACM SIGMOD International Conference on Management of Data (SIGMOD'09), Providence, RI, 2009
  26. L. Bouganim, B. Jonsson and P. Bonnet. uFLIP: Understanding Flash IO Patterns. CIDR 2009.
  27. V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. USENIX 2007.
  28. G.-J. Kim, S.-C. Baek, H.-S. Lee, et al. Lgedbms: A Small Dbms for Embedded System with Flash Memory. VLDB '06: Proceedings of the 32nd international conference on Very large data bases. Seoul, Korea . 2006: 1255 - 1258
  29. S.-W. Lee, B. Moon, C. Park, et al. A Case for Flash Memory Ssd in Enterprise Database Applications. SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Vancouver, Canada . 2008. New York, NY, USA, 2008:1075 - 1086
  30. Lawder J K, King P J H. Using Space-filling Curves for Multi-Dimensional Indexing. In Proc. BNCOD 17, Lectures Notes in Computer Science, Springer 2000, 1832: 20-35
  31. Moon B, Jagadish H V, Faloutsos C, Saltz J H. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. IEEE Transactions on Knowledge and Data Engineering, Volume: 13, Issue: 1: On page(s): 124-141
  32. [http://en.wikipedia.org/wiki/Hilbert\\_curve](http://en.wikipedia.org/wiki/Hilbert_curve)
  33. S. Saltenis, C. Jensen. Indexing of Moving Objects for Location-based Services. Proceedings. 18th International Conference on Data Engineering, 2002: 463 - 4729
  34. P. K. Agarwal, L. Arge, J. Erickson. Indexing moving points. Symposium on Principles of Database Systems. Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. Dallas, Texas, United States. Pages: 175 - 186

35. Mindaugas Pelanis, Simonas Saltenis, Christian S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Transactions on Database Systems (TODS)*. Volume 31, Issue 1 (March 2006). Pages: 255 - 298
36. J Nievergelt, H Hinterberger, KC Sevcik. Grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, Volume 9 , Issue 1 (March 1984) :Pages: 38 - 71
37. C. S. Jensen, D. Lin, B. C. Ooi. Query and Update Efficient B+-tree Based Indexing of Moving Objects. *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. Toronto, Canada. VLDB Endowment . 2004: 768 - 779
38. ML Yiu, Y Tao, N Mamoulis. The B dual-Tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal — The International Journal on Very Large Data Bases*. Volume 17 , Issue 3 (May 2008). Pages: 379 - 400
39. S. Chen, B. C. Ooi, K.-L. Tan, et al. St<sup>2</sup>b-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects. *SIGMOD Conference*. Vancouver, Canada. 2008:29 - 42
40. N Beckmann, HP Kriegel, R Schneider, B Seeger. The R\*-tree: an efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. Atlantic City, New Jersey, United States. Pages: 322 - 331
41. RA Finkel, JL Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, Volume 4, Number 1, 1974. Page 1-9
42. Y Tao, D Papadias, J Sun .The TPR\*-tree: An optimized spatio-temporal access method for predictive queries. *Proceedings of the 29th international conference on Very large data bases*. Berlin, Germany. 2003: Pages: 790 - 801
43. 刘晓军. 一种高效的移动对象位置索引机制的研究. 山东大学硕士学位论文. 2007
44. 张文杰. 移动对象全时态索引结构与查询处理技术研究. 哈尔滨工业大学硕士学位论文. 2006
45. Wu, K.-L.; Chen, S.-K.; Yu, P.S., "Incremental Processing of Continual Range Queries over Moving Objects," *Knowledge and Data Engineering*, IEEE

Transactions on , vol.18, no.11, pp.1560-1575

## 哈尔滨工业大学硕士学位论文原创性声明

本人郑重声明：此处所提交的硕士学位论文《数据频繁更新环境下的 Flash 存储管理问题研究》，是本人在导师指导下，在哈尔滨工业大学攻读硕士学位期间独立进行研究工作所取得的成果。据本人所知，论文中除已注明部分外不包含他人已发表或撰写过的研究成果。对本文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。本声明的法律结果将完全由本人承担。



作者签字：

日期：2009 年 6 月 29 日

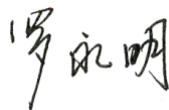
## 哈尔滨工业大学硕士学位论文使用授权书

《数据频繁更新环境下的 Flash 存储管理问题研究》系本人在哈尔滨工业大学攻读硕士学位期间在导师指导下完成的硕士学位论文。本论文的研究成果归哈尔滨工业大学所有，本论文的研究内容不得以其它单位的名义发表。本人完全了解哈尔滨工业大学关于保存、使用学位论文的规定，同意学校保留并向有关部门送交论文的复印件和电子版本，允许论文被查阅和借阅，同意学校将论文加入《中国优秀博硕士学位论文全文数据库》和编入《中国知识资源总库》。本人授权哈尔滨工业大学，可以采用影印、缩印或其他复制手段保存论文，可以公布论文的全部或部分内容。

本学位论文属于（请在以下相应方框内打“√”）：

保密 ，在 \_\_\_\_\_ 年解密后适用本授权书

不保密



作者签名：

日期：2009 年 6 月 29 日



导师签名：

日期：2009 年 6 月 29 日

## 致谢

感谢我的导师李建中教授几年来对我的悉心培养。李建中老师高尚的品德，诚实的学风，扎实的理论功底，前瞻的学术眼光永远是我学习的榜样。您对待学术的勤奋和认真使我汗颜，您为学术献身的精神使我深深感动。从师于您是我人生一大幸事，希望您身体更加健康，生活中充满欢乐。

感谢张炜老师对我的指导和帮助。在您的指导下，我对分析和解决问题的整个过程有了一个新的认识，使我在完成论文的过程中受益匪浅。这许多的指导和帮助，为我的毕业设计的完成奠定了基础。

感谢高宏老师、王宏志老师对我学习和生活的热切关怀。在我遇到困难的时候，你们的建议和行动给了我直接的支持与鼓励。还要感谢实验室的石胜飞老师、骆吉洲老师，他们也都给予了我很多帮助。感谢孟啸同学与我进行探讨，这些探讨拓宽了我的思路。还要感谢实验室所有帮助过我、跟我讨论问题的老师和同学们，在此就不一一列举了。

衷心感谢两年来陪我度过研究生生活的数据与知识工程研究中心 07 级硕士同学们，你们的笑容和鼓励伴我越过了一个又一个困难，欣赏了一路的人生美景。你们每个人身上的闪光都是我努力的方向。

最后，深深感谢多年来抚育我成长的父母。你们是最坚强的后盾，是我实现梦想的力量源泉。